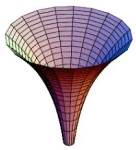


SKSurrogate Documentation

Release 0.2.0

Mehdi Ghasemi

Sep 29, 2023

Contents:

CHAPTER 1

Introduction

`SKSurrogate` is a suite of tools focused on designing and tuning machine learning pipelines and track the evolution of simple modeling tasks. `SKSurrogate` is designed to employ Surrogate Optimization technique in a flexible way which enjoys from the wealth of existing well-known python tools. It provides a ui to perform surrogate optimization on general functions and has a familiar ui to perform hyperparameter tuning for scikit-learn compatible models. Moreover, `SKSurrogate` uses surrogate optimization and evolutionary optimization algorithms to construct and tune complex pipelines (automl) based on a set of given (scikit-learn compatible) models, like TPOT does.

Note that automl part is not designed to result in outstanding results in a few minutes, rather it needs hours if not days to come up with a good result. There are various methods that speeds up the pipeline design/optimization process such as selecting faster surrogates, fewer number of steps and evolutionary optimization settings.

The evolutionary optimization module is designed to be very flexible and can be modified to perform evolutionary optimization on any given evolutionary compatible problem.

1.1 Dependencies

- NumPy,
- scipy,
- pandas,
- matplotlib,
- scikit-learn,
- ELI5,
- SALib,
- peewee.

1.2 Download

SKSurrogate can be obtained from https://github.com/mghasemi/sksurrogate.

1.3 Installation

To install *SKSurrogate*, run the following in terminal:

```
sudo python setup.py install
```

1.4 Documentation

The documentation is produced by Sphinx and is intended to cover code usage as well as a bit of theory to explain each method briefly. For more details refer to the documentation at sksurrogate.rtfd.io.

1.5 License

This code is distributed under MIT license:

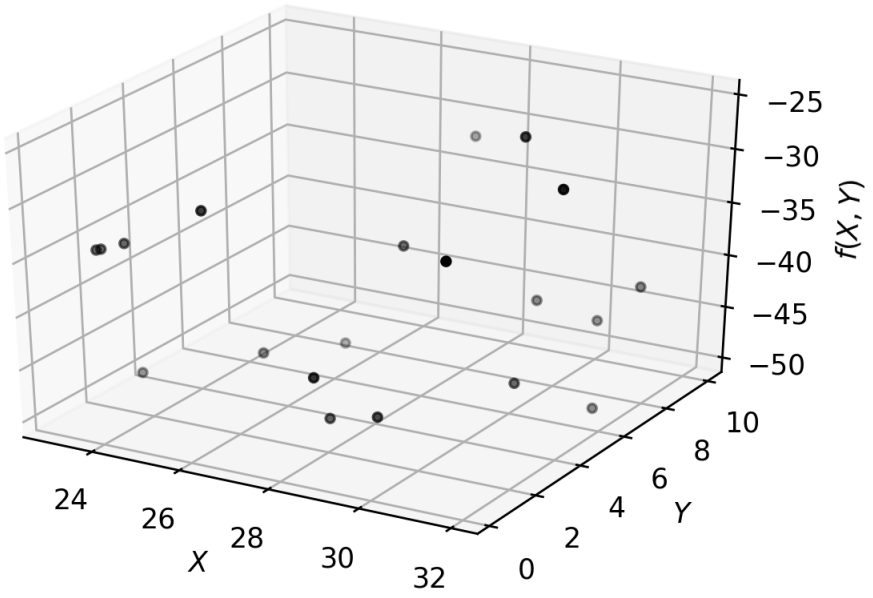
1.5.1 MIT License

CHAPTER 2

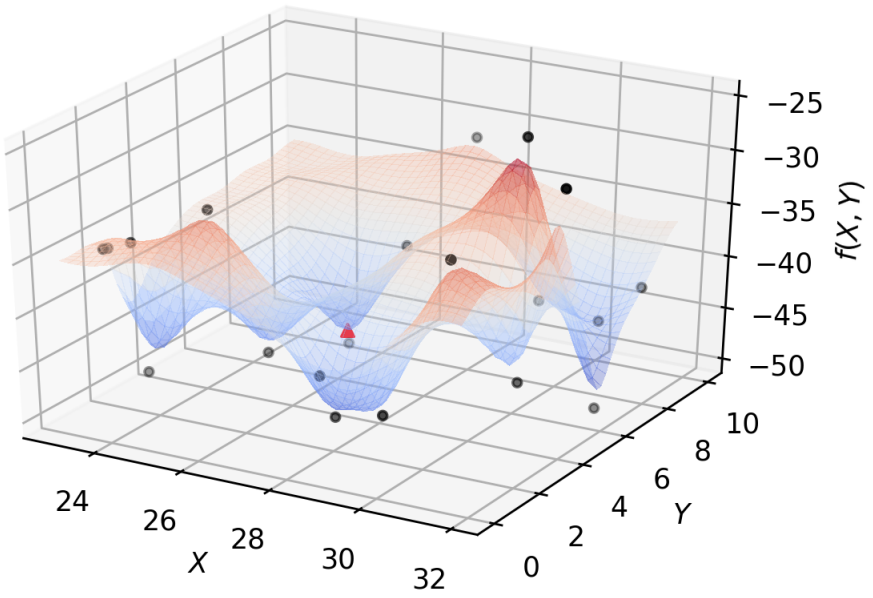
Surrogate-Based Optimization

Surrogate-based optimization represents a class of optimization methodologies that make use of surrogate modeling techniques to quickly find the local or global optima. It provides us a novel optimization framework in which the conventional optimization algorithms, e.g. gradient-based or evolutionary algorithms are used for sub-optimization.

For optimization problems, surrogate models can be regarded as approximation models for the cost function and state function, which are built from sampled data obtained by randomly probing the design space. Once the surrogate models are built, an optimization algorithm can be used to search the new candidate, based on the surrogate models, that is most likely to be the optimum. Since the prediction with a surrogate model is generally much more efficient than that with a numerical analysis code, the computational cost associated with the search based on the surrogate models is generally negligible. Surrogate modeling is referred to as a technique that makes use of the sampled data to build surrogate models, which are sufficient to predict the output of an expensive function at untried points in the feasibility space. Thus, how to choose sample points, how to build surrogate models, and how to evaluate the accuracy of surrogate models are key issues for surrogate optimization.



Fit a regressor and find a (local) minimum:



The following algorithm summarizes implemented Surrogate-Based Optimization for an expensive function f that can

only be evaluated *MaxIter* times with a minimum *MinEvals* random evaluations.

Note: set $E = \emptyset$, $NumIter = 0$

while $NumIter < MaxIter$ do

sample a random point x_0 from feasible space

if $\#(E) < MinEvals$ **then** update $E = E \cup \{(x_0, f(x_0))\}$

else find a surrogate $\hat{f}$ that fits the points in E

find the minimum x_* of $\hat{f}$ using x_0 as the initial point (if required)

evaluate f at x_* and update $E = E \cup \{(x_*, f(x_*))\}$

update $NumIter = NumIter + 1$

return the pair $(x, y) \in E$ with lowest found value for y as the approximation for the minimum of f

---

Clearly, there are various methods to accomplish some of the steps in the above algorithm like how to sample a new point x_0, how to find the surrogate $\hat{f}$, how to minimize and how to decide when we wish to use the surrogate.

2.1 Sampling

Three different sampling methods have been implemented and the *SurrogateSearch* class accepts user defined sampling classes that have a certain signature.

2.1.1 CompactSample

This class samples a random point from the feasibility set.

2.1.2 BoxSample

This class samples a random point from a cube centered around a given point (usually the last point f was evaluated on) with a given length. It makes sure that the sample belongs to the feasibility set.

The length of the edges of the cube can be provided by setting *init_radius* (default: 2.)

A contraction ratio can be provided by setting *contraction* (should be bigger than 0 and less than 1) to shrink the volume of the cube to assure fast convergence to a (local) optima (default value: 0.9).

2.1.3 SphereSample

This class samples a random point from a sphere centered around a given point (usually the last point f was evaluated on) with a given radius. It makes sure that the sample belongs to the feasibility set.

The radius of the sphere can be provided by setting *init_radius* (default: 2.)

A contraction ratio can be provided by setting *contraction* (should be bigger than 0 and less than 1) to shrink the radius of the sphere to assure fast convergence to a (local) optima (default value: 0.9).

Note: The sampling method can be passed to an instance of *SurrogateSearch* via *sampling* parameter. Along with *sampling* class, *radius*, *contraction*, *ineq*, and *bounds* may be provided to be used by the sampling class. *ineq* is a list

of callables which represent the constraints. *bounds* is a list of tuples of real numbers representing the bounds on each variable.

Tip: A user-defined sampling class should follow the following structure:

```
class UserSample(object):
    def __init__(self, **kwargs):
        pass

    def check_constraints(self, point):
        """
        Checks constraints on the sample if provided;
        `point` is the candidate to be checked;
        should return a `boolean` True or False for if all constraints hold or not.
        """
        pass

    def sample(self, centre, cntrctn=1.):
        """
        Samples a point out of an sphere centered at `centre`;

        `centre` is a `numpy.array` the center of the sphere;
        `cntrctn` is a  `float` customized contraction factor
        returns a `numpy.array` the new sample
        """
        pass
```

2.2 Surrogate models

By default, *SurrogateSearch* uses a polynomial surface of degree 3 to approximate f based on existing data and will be updated on each iteration where a new piece of information a bout f is found. Typically, any regressor inherited from *RegressorMixin* that implements a *fit* and a *predict* method can be used.

2.3 Optimizer

A *scipy* optimizer can be used to find a minimum of the surrogate at each iteration. Note that if *ineqs* is not *None*, then most of scipy optimizers can not be used. The optimizers that work well with constraints include 'SLSQP' and 'COBYLA'.

An alternative for the scipy optimizer is 'Optimithon'.

CHAPTER 3

Hyperparameter Optimization

Hyperparameter Optimization in machine learning with respect to a given performance measure (e.g., accuracy, f1, auc, ...) usually is a computationally expensive task which fits within the scope of surrogate optimization technique. In fact this is the main reason that the project exists. Therefore, there is a special class designed for hyperparameter optimization of machine learning methods that follow the schema of the very popular machine learning library *scikit-learn*. The class *structsearch.SurrogateRandomCV* is a substitute for scikit-learn's *GridSearchCV* or *RandomizedSearchCV*. An isntance of *SurrogateRandomCV* takes an estimator like *GridSearchCV* and *RandomizedSearchCV* and a *params* parameter, like *param_grid* or *param_distributions*, which determines (ranges of) the values each argument of the estimator can take over. The difference is that not only it accepts discrete list of values for each parameter, it also accepts ranges of integers and real numbers too. The *params* is a dictionary whose keys are the estimator's arguments and their values are objects of the followin types:

- *Real(a, b)*: an interval of real numbers between a and b;
- *Integer(a, b)*: an interval of integer numbers between a and b;
- *Categorical(list)*: a list consists of discrete values;
- *HDReal(a, b)*: An n dimensional box of real numbers corresponding to the classification groups (e.g. *class_weight*). a is the tuple of lower bounds and b is the tuple of upper bounds.

Example The following code searches for the best values for a *SVC*:

```
from sklearn.svm import SVC
from SKSurrogate import *
clf = SVC()
params = {'C': Real(1.e-5, 10),
          'kernel': Categorical(['poly', 'rbf']),
          'degree': Integer(1, 4),
          'gamma': Real(1.e-5, 10),
          'class_weight': HDReal((1.e-3, 1.e-3), (10., 10.))}
srch = SurrogateRandomCV(clf, params)
srch.fit(X, y)
print(srch.best_estimator_)
```

Note: It is worth mentioning that using a Gaussian Process Regression as the regressor simulates a particular variation

of the optimization method known as Bayesian Optimization method.

Bayesian Optimization is the method employed by the popular package skopt. The ui implemented for `SurrogateRandomCV` is very much similar to the one of `skopt.BayesSearchCV`, so one could use the same code for both given that the imports are carefully done.

Tip: The class `SurrogateRandomCV` works with intervals to handel the hyperparameters and the current sampling classes do not impose extra constraints on `SurrogateSearch` other than ranges for parameters, alternative *scipy.minimize* solvers can be used as well, such as *L-BFGS-B*, *TNC*, *SLSQP*.

CHAPTER 4

Hilbert Spaces

4.1 Orthonormal system of functions

Let *X* be a topological space and μ be a finite Borel measure on *X*. The bilinear function $\langle \cdot, \cdot \rangle$ defined on $L_2(X, \mu)$ as $\langle f, g \rangle = \int_X f g d\mu$ is an inner product which turns $L_2(X, \mu)$ into a Hilbert space.

Let us denote the family of all continuous real valued functions on a non-empty compact space *X* by $\mathrm{C}(X)$. Suppose that among elements of $\mathrm{C}(X)$, a subfamily *A* of functions are of particular interest. Suppose that *A* is a subalgebra of $\mathrm{C}(X)$ containing constants. We say that an element $f \in \mathrm{C}(X)$ can be approximated by elements of *A*, if for every $\epsilon > 0$, there exists $p \in A$ such that $|f(x) - p(x)| < \epsilon$ for every $x \in X$. The following classical results guarantees when every $f \in \mathrm{C}(X)$ can be approximated by elements of *A*.

Let $(V, \langle \cdot, \cdot \rangle)$ be an inner product space with $\|v\|_2 = \langle v, v \rangle^{\frac{1}{2}}$. A basis $\{v_\alpha\}_{\alpha \in I}$ is called an orthonormal basis for *V* if $\langle v_\alpha, v_\beta \rangle = \delta_{\alpha\beta}$, where $\delta_{\alpha\beta} = 1$ if and only if $\alpha = \beta$ and is equal to *0* otherwise. Every given set of linearly independent vectors can be turned into a set of orthonormal vectors that spans the same sub vector space as the original. The following well-known result gives an algorithm for producing such orthonormal from a set of linearly independent vectors:

Note: Gram–Schmidt

Let $(V, \langle \cdot, \cdot \rangle)$ be an inner product space. Suppose $\{v_i\}_{i=1}^n$ is a set of linearly independent vectors in *V*. Let

$$u_1 := \frac{v_1}{\|v_1\|_2}$$

and (inductively) let

$$w_k := v_k - \sum_{i=1}^{k-1} \langle v_k, u_i \rangle u_i \text{ and } u_k := \frac{w_k}{\|w_k\|_2}.$$

Then $\{u_i\}_{i=1}^n$ is an orthonormal collection, and for each *k*,

$$span\{u_1, u_2, \cdots, u_k\} = span\{v_1, v_2, \cdots, v_k\}.$$

Note that in the above note, we can even assume that $n = \infty$.

Let $B = \{v_1, v_2, \dots\}$ be an ordered basis for $(V, \langle \cdot, \cdot \rangle)$. For any given vector $w \in V$ and any initial segment of *B*, say $B_n = \{v_1, \dots, v_n\}$, there exists a unique $v \in \text{span}(B_n)$ such that $\|w - v\|_2$ is the minimum:

Note: Let $w \in V$ and *B* a finite orthonormal set of vectors (not necessarily a basis). Then for $v = \sum_{u \in B} \langle u, w \rangle u$

$$\|w - v\|_2 = \min_{z \in \text{span}(B)} \|w - z\|_2.$$

Now, let μ be a finite measure on *X* and for $f, g \in \text{C}(X)$ define $\langle f, g \rangle = \int_X f g d\mu$. This defines an inner product on the space of functions. The norm induced by the inner product is denoted by $\| \cdot \|_2$. It is evident that

$$\|f\|_2 \leq \|f\|_\infty \mu(X), \ \forall f \in \text{C}(X),$$

which implies that any good approximation in $\| \cdot \|_\infty$ gives a good $\| \cdot \|_2$-approximation. But generally, our interest is the other way around. Employing Gram-Schmidt procedure, we can find $\| \cdot \|_2$ within any desired accuracy, but this does not guarantee a good $\| \cdot \|_\infty$-approximation. The situation is favorable in finite dimensional case. Take $B = \{p_1, \dots, p_n\} \subset \text{C}(X)$ and $f \in \text{C}(X)$, then there exists $K_f > 0$ such that for every $g \in \text{span}(B \cup \{f\})$,

$$K_f \|g\|_\infty \leq \|g\|_2 \leq \|g\|_\infty \mu(X).$$

Since *X* is assumed to be compact, $\text{C}(X)$ is separable, i.e., $\text{C}(X)$ admits a countable dimensional dense subvector space (e.g. polynomials for when *X* is a closed, bounded interval). Thus for every $f \in \text{C}(X)$ and every $\epsilon > 0$ one can find a big enough finite *B*, such that the above inequality holds. In other words, good enough $\| \cdot \|_2$-approximations of f give good $\| \cdot \|_\infty$-approximations, as desired.

Example. Polynomial regression on 2-dimensional random data:

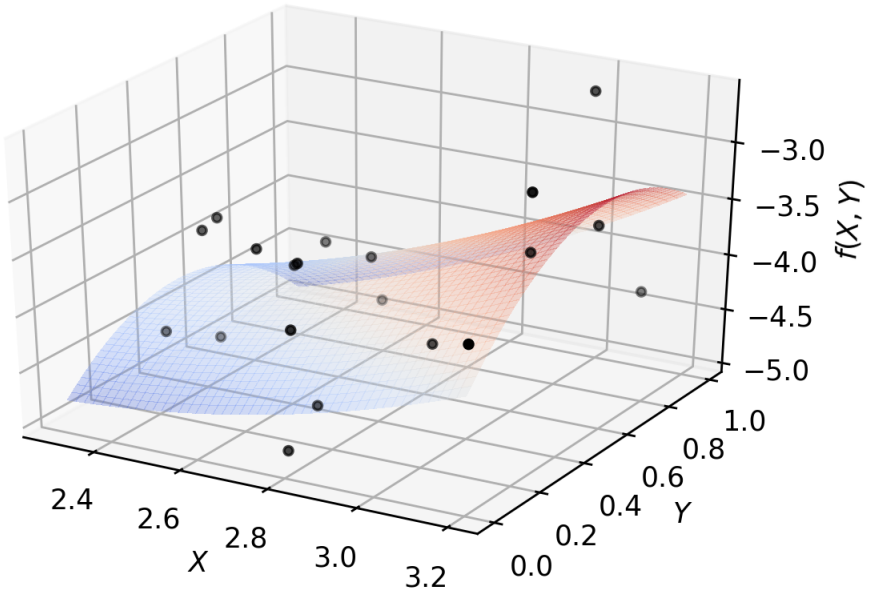
```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import matplotlib.pyplot as plt
import numpy as np
from SKSurrogate.NpyProximation import HilbertRegressor, FunctionBasis
def randrange(n, vmin, vmax):
        '''
        Helper function to make an array of random numbers having shape (n, )
        with each number distributed Uniform(vmin, vmax).
        '''
        return (vmax - vmin)*np.random.rand(n) + vmin

# degree of polynomials
deg = 2
FB = FunctionBasis()
B = FB.Poly(2, deg)
# initiate regressor
regressor = HilbertRegressor(base=B)
# number of random points
n = 20
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for c, m, zlow, zhigh in [('k', 'o', -5, -2.5)]:
        xs = randrange(n, 2.3, 3.2)
        ys = randrange(n, 0, 1.0)
```

(continues on next page)

(continued from previous page)

```
        zs = randrange(n, zlow, zhigh)
        ax.scatter(xs, ys, zs, c=c, s=10, marker=m)
ax.set_xlabel('$X$')
ax.set_ylabel('$Y$')
ax.set_zlabel('$f(X,Y)$')
X = np.array([np.array((xs[_], ys[_])) for _ in range(n)])
y = np.array([np.array((zs[_],)) for _ in range(n)])
X_ = np.arange(2.3, 3.2, 0.02)
Y_ = np.arange(0, 1.0, 0.02)
_X, _Y = np.meshgrid(X_, Y_)
# fit the regressor
regressor.fit(X, y)
# prepare the plot
Z = []
for idx in range(_X.shape[0]):
        _X_ = _X[idx]
        _Y_ = _Y[idx]
        _Z_ = []
        for jdx in range(_X.shape[1]):
                t = np.array([np.array([_X_[jdx], _Y_[jdx]])])
                _Z_.append(regressor.predict(t)[0])
        Z.append(np.array(_Z_))
Z = np.array(Z)
surf = ax.plot_surface(_X, _Y, Z, cmap=cm.coolwarm, linewidth=0, antialiased=False,␣
↪alpha=.3)
```



CHAPTER 5

Sensitivity Analysis

This module could be easily taken off of this library. But one could take advantage of sensitivity analysis to reduce the complexity of expensive models. Moreover, observations in practice shows significant gains in performance by employing data preprocessing based on sensitivity analysis.

Sensitivity analysis is defined as the study of how the uncertainty in the output of a model can be apportioned to sources of uncertainty in inputs.

Given a model $y = f(x_1, \ldots, x_n)$, the following are standard sensitivity measures quantifying sensitivity of the model with respect to $x_1, \ldots, x_n$:

5.1 Morris

The **Morris** method facilitates a global sensitivity analysis by making a number of local changes at different points of the possible range of input values. The following quantities are usually measured regarding Morris method:

$$\mu_i = \int \frac{\partial f}{\partial x_i} dx_1 \cdots dx_n,$$
$$\mu_i^* = \int |\frac{\partial f}{\partial x_i}| dx_1 \cdots dx_n,$$
$$\sigma_i = Var(\frac{\partial f}{\partial x_i}).$$

Generally, μ^* is used to detect input factors with an important overall influence on the output. σ is used to detect factors involved in interaction with other factors or whose effect is non-linear.

5.2 Sobol

The **Sobol** method (aka variance-based sensitivity analysis) works by decomposing the variance of the output of the model into fractions which can be attributed to inputs or sets of inputs. The first-order indices are defined as:

$$S_i = \frac{D_i(y)}{Var(y)}, \quad S_{ij} = \frac{D_{ij}(y)}{Var(y)}, \ldots$$

where

$$D_i(y) = Var_{x_i}(E_{x_{-i}}(y|x_i)),\ D_{ij}(y) = Var_{x_{ij}}(E_{x_{-ij}}(y|x_i, x_j)) - (D_i(y) + D_j(y)), \ldots,$$

and the total-effect index:

$$S_{T_i} = \frac{E_{x_{-i}}(Var_{x_i}(y|x_{-i}))}{Var(y)} = 1 - \frac{Var_{x_{-i}}(E_{x_i}(y|x_{-i}))}{Var(y)}.$$

5.3 Moment-Independent δ Index

Let $g_Y(y)$ be the distribution of the values of y and denote by $g_{Y|x_i}(y)$ the distribution of values of y when the value of x_i is fixed. Let $s(x_i) = \int |g_Y(y) - g_{Y|x_i}(y)|dy$, then the delta index of x_i is defined as:

$$\delta_i = \frac{1}{2} \int s(x_i) g_{x_i} dx_i,$$

where $g_{x_i}(x_i)$ is the distribution of the values of x_i.

Note: The class *SensAprx* acts as a scikit-learn wrapper as a transformer based on the sensitivity analysis library SALib.

It accepts a scikit-learn compatible regressor at initiation, fits the regressor on the X, y arguments of *SensAprx.fit* and performs sensitivity analysis on the regressor.

- The type of analysis can be determined at initiation by choosing *method* among *['sobol', 'morris', 'delta-mmnt']* (default: 'sobol').
- After calling the *fit* method, coefficients are stored in *SensAprx.weights_*.
- After calling *SensAprx.fit* by calling *SensAprx.transform(X)* selects the top *n* features where *n* is given at initiation through *n_features_to_select*.
- It is easier to do sensitivity analysis on functions using SALib's ui, but if one prefers using scikit-learn's wrapper, then the function should be modified to resemble a scikit-learn regressor which simply ignores training data.

CHAPTER 6

Eliminate features based on Pearson correlation

The Pearson correlation is widely used to eliminate some of highly correlated features to reduce the number of features. Surprisingly, there is no *scikit-learn* compatible code implementing feature selection according to a given correlation threshold (at the time publishing this library).

Although, it sound like an easy task to do, it is not clear how to select a minimal set of features with low correlation and which ones can be safely excluded.

SKSurrogate.sensapprx.CorrelationThreshold implements the following algorithm to select a minimal set of features with correlation below a given threshold:

Note: **Input:** the set of all variables V, a positive threshold t

Output a subset $W \subseteq V$ where $\forall a, b \in W \quad |corr(a, b)| < t$

set $P :=$ the set of all pairs $(a, b) \in V \times V$ where $|corr(a, b)| \geq t$

set $W := \{v \in V : \forall x \in V | corr(v, x)| < t\}$

while $P \neq \emptyset$ **do:** form the undirected graph $G = G(V \setminus W, P)$ find a node $w \in V \setminus W$ with highest degree update $W := W \cup \{w\}$ remove all pairs from P involving w

return W

The above procedure selects those features which has high (positive or negative) correlation with higher number of other features and omits the other features. Repeats this process until no more pair with high correlation remains.

CHAPTER 7

Evolutionary Optimization Algorithms

An Evolutionary Algorithm (EA) uses mechanisms inspired by biological evolution, such as *reproduction*, *mutation*, *recombination* and *selection*. Candidate solutions to the optimization problem play the role of individuals in population and the fitness function determines the quality of the solution. Evolution of the population then takes place after the repeated application of the above operators.

7.1 A General Evolutionary Optimization Algorithm

A typical evolutionary optimization algorithm consists of the following four operations: reproduction, mutation, recombination and selection and elitism. These operations are usually performed in the following order with many variations:

Note: Randomly generate a population $\longrightarrow$ Parents

While not (Termination criterion):

Calculate the fitness of all Parents

Best E Parents $\longrightarrow$ Elites

$\emptyset \longrightarrow$ Children

While #(Children) < #(Parents):

Use fitness to probabilistically select pairs of Parents for recombination

Recombine (Mate) Parents to create Children c_1, c_2

$Children \cup \{c_1, c_2\}$

Randomly mutate Children

$Children \cup Elites \longrightarrow Parents$

Best N Parents $\longrightarrow$ Parents

The above algorithm is implemented in *eoa.EOA*. An instance of this class requires two mandatory inputs: (1) *population* which is a list, consisting of all possible individuals, and (2) *fitness* which is a function that accepts a list of individuals and returns an *OrderedDict* with individuals as its keys and their fitness as values.

The individuals are assumed to be python tuples, typically generated by *aml.Words*. An instance of *aml.Words* requires a list of alphabets *letters*. If one requires a restiction on those letters that can appear at the end of a tuple or at the begining (as well as other positions in a tuple) then they should be indicated as parameters *first* and *last*. The parameter *repeat* indicated whether consecutive appearance of letters is allowed or not. The method *Words.Generate* takes a parameter as the length of tuples and produces the list of all legitimate tuples of a given length.

The first step in the above algorithm is selecting a random subset of the population as initial parents. One may consider various factors to impact on the initial parents such as the length of individuals, initial letters, etc. The default selection strategy for *eoa.EOA* is simply selecting a random population of a certain size. This behaviour can be changed by specifying the *init_pop* parameter of the *EOA* class (default *eoa.UniformRand*). *init_pop* accepts user-defined classes as input, but it assumes that the user-defined class implements a *__call__* method whose first input should be the parent instance of the *EOA* class, which iplies that the user-defined class has access to all properties of the parent *EOA* instance. This remains the default coding MO for other user-defined classes that will be accepted by *EOA*. The *__call__* should return an *OrderedDict* whose keys are individuals from population and their values are their fitness, that may have been calculated before and stores in *EOA.evals* which itself is an *OrderedDict* of similar type.

The *eoa.EOA* requires a termination criterion too. The defaulkt tewrmination criterion is set to be reaching a certain number of generations. This can be modified by specifying a user-defined termination class. A user-defined termination criterion is a class that implements *__call__* which accepts the parent instance of the *EOA*. The default termination criterion is *eoa.MaxGenTermination* which checks whether the *max_generations* is reached or not. The *max_generations* can be specified on initialization of the *EOA* instance by setting the *max_generations*. The *__call__* method should return a boolean indicating the satisfaction of the termination criterion.

To Select mating parents and produce their children, *eoa.EOA* uses a class provided to the instance of the *EOA* via *recomb* parameter. The default class is *eoa.UniformCrossover*. *UniformCrossover* uses the parents fitness to bias toward those that are most fit for mating and selects them probabilistically, the higher the fitness value, the higher the chance of finding a mate. After the pairing procedure, we then recombine the pairs (p_1, p_2) to produce two children. The default procedure, chooses a random integer $1 \leq l \leq \max(\#(p_1), \#(p_2))$, put identity functions at the beginning of the shorter parent to make their length equal. Then cut them at l^{th} position and combine the initial part of p_1 with later part of p_2 and initial part of p_2 with later part of p_1 to produce children.

p_1:	g_1	$\ldots$	g_{l-1}	g_l	$\ldots$	g_n
p_2:	h_1	$\ldots$	h_{l-1}	h_l	$\ldots$	h_n

$\Downarrow$

c_1:	g_1	$\ldots$	g_{l-1}	h_l	$\ldots$	h_n
c_2:	h_1	$\ldots$	h_{l-1}	g_l	$\ldots$	g_n

If $\#(p1) = \#(p_2) = 1$, then *UniformCrossover* simply puts $c_1 = (p_1, p_2)$ and $c_2 = (p_2, p_1)$.

Note that a user-defined mating class needs to implement a *__call__* method which accepts the parent *EOA* instance and sets its *children* method as an *OrderedDict* whose keys are the individuals and their values are their fitness. It is recommended to sort the *children* dictionary by its values before returning.

The next step is mutation. This process assures that even isolated individuals in the population has a chance to be explored. The default behaviour of *eoa.EOA* is implemented as *eoa.Mutation* which uses *mutation_prob* (default = 0.05) to randomly changes entities of each child. More accurately, each element of a child, will be changed into another (legitimate) element with probability *mutation_prob*. Again, the default behaviour can be modified by specifying a user-defined class which implement a *__call__* method accepting the parent instance of *EOA* and modifies its *.children* dictionary and their fitness.

The last step in this implementation is called elitism. The purpose of the elitism is to keep those parents that are better fit compare to some of the children among the next generation parents. The default behaviour of *eoa.EOA* is implemented as *eoa.Elites* and can be modified by assigning a user-defined class to *elitism* parameter of *eoa.EOA*. The user-defined class requires to implement a *__call__* method that accepts the parent *EOA* instance and modifies the parent's *children*.

7.1.1 Example

The following is a synthetic example on a small set of letters and a fitness based on ASCI code and length of individuals:

```
# prepare the whole population
P = Words(['a', 'b', 'c', 'd', '1', '2'], last=['1', '2', '3'], repeat=True)
Pop = P.Generate(1) + P.Generate(2) + P.Generate(3) + P.Generate(4) +
      P.Generate(5) + P.Generate(6) + P.Generate(7)
# initiate the EOA instance
gen = EOA(population=Pop, fitness=sfit, num_parents=100, mutation_prob=.1, term_
↪genes=['1', '2'])
# run the EOA
gen()
# get the most fit individual found and print
best = next(reversed(tst.children))
print(best, tst.children[best])
```

produces the following output:

```
100%|##########| 50/50 [00:00<00:00, 769.22it/s]
('1', '1', '1', '1', '1', '1', '1') 10.149999999999999
```

CHAPTER 8

Optimized Pipeline Detector

In this part, we speculate about possibility of searching for optimized pipelines that perform some preprocessing, sensitivity analysis, and composing estimators to achieve an optimum performance measure, based on a pre-determined set of operators and estimators, all inherited from scikit-learn's base classes.

Suppose that we have some data transformers $\mathbb{T} = \{T_1, \ldots, T_k\}$ and some estimators $\mathbb{E} = \{E_1, \ldots, E_m\}$ and willing to find a composition $P = F_1 \circ F_2 \circ \cdots \circ F_n$ where $F_1 \in \mathbb{E}$ and $F_i \in \mathbb{T} \cup \mathbb{E}$ for $i = 2, \ldots, n$ and the composition is optimal with respect to a given performance measure. Each estimator/transformer may accept a number of parameters, discrete or continuous. Note that there are $m \times (m+k)^{n-1}$ different combinations based on $\mathbb{T}$ and $\mathbb{E}$. So, the number of possible pipelines grows exponentially as the number of building blocks increase. Now, if we want to examine all possible combinations of at most N estimator/transformer, the domain would be of the form

$$\mathbb{U} = \bigcup_{n=1}^{N} \mathbb{E} \times (\mathbb{E} \cup \mathbb{T})^{n-1}.$$

each element of $\mathbb{U}$ corresponds to infinitely many functions as the set of acceptable hyperparameters for each one is potentially infinite. Suppose that $P \in \mathbb{U}$ and $\tilde{x}$ is the set of its parameters. We use surrogate optimization to find

$$\tilde{x}_P = \text{argmax}_{\tilde{x}} \mu(P(\tilde{x})(X)),$$

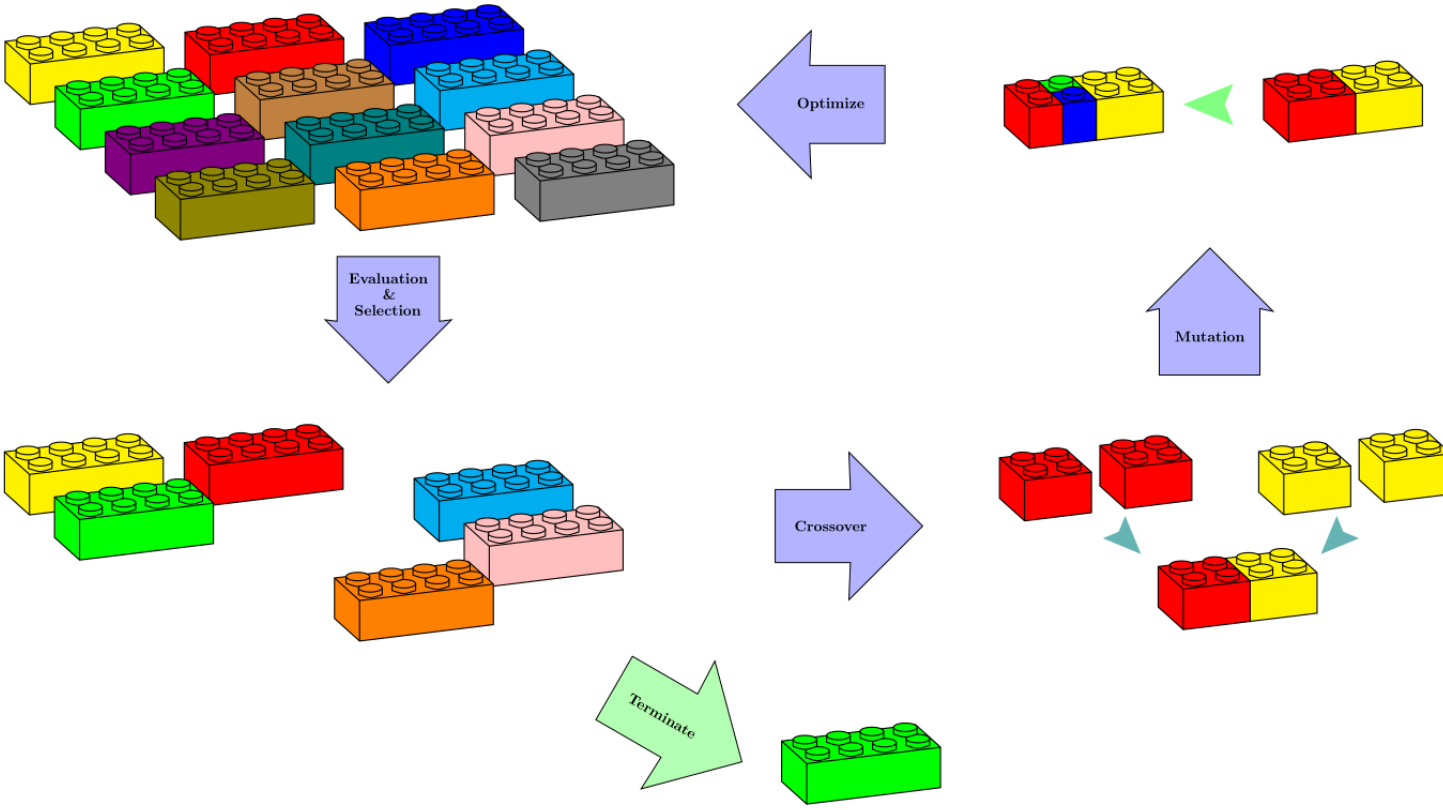
(or argmax depending on the task) where X is the test set, μ is the performance and deal with the elements of

$$\tilde{\mathbb{U}} = \{P(\tilde{x}_P) : P \in \mathbb{U}\},$$

that are already optimized with respect to their hyperparameters. This reduces the optimized pipeline detection to searching $\tilde{\mathbb{U}}$ to find an optimum pipeline. This is still a very heavy task to accomplish given the number of elements in $\mathbb{U}$ and the computational intensity of a surrogate optimization. Fortunately, the format of the elements of $\mathbb{U}$ is very much suggestive and demands for a genetic algorithm to reach an optima.

The *AML* class accepts a set of estimators and transformers, dictionaries of their parameters that can be changed, and searches the space of possible pipelines either exhaustively or according to an evolutionary set up to find an optimum pipeline.

Think of machine learning pipelines as Lego blocks with each stud representing a step in the pipeline. The following diagram summarizes the whole evolutionary process implemented to detect the optimal pipeline:



Example 1: The following is a classification based on sk-rebate data:

```
# Find an optimum classification pipeline

import pandas as pd
import numpy as np
from sklearn.model_selection import RandomizedSearchCV
from sklearn.kernel_ridge import KernelRidge
from sklearn.gaussian_process.kernels import Matern, Sum, ExpSineSquared
from SKSurrogate import *

param_grid_krr = {
    "alpha": np.logspace(-4, 0, 5),
    "kernel": [
        Sum(Matern(), ExpSineSquared(l, p))
        for l in np.logspace(-2, 2, 10)
        for p in np.logspace(0, 2, 10)
    ],
}
regressor = RandomizedSearchCV(
    KernelRidge(), param_distributions=param_grid_krr, n_iter=5, cv=2
)

config = {
    # Classifiers
    "sklearn.naive_bayes.GaussianNB": {"var_smoothing": Real(1.0e-9, 2.0e-1)},
    "sklearn.linear_model.LogisticRegression": {
        "penalty": Categorical(["l1", "l2"]),
        "C": Real(1.0e-6, 10.0),
        "class_weight": HDReal((1.0e-5, 1.0e-5), (20.0, 20.0))
```

(continues on next page)

(continued from previous page)

```
        # 'dual': Categorical([True, False])
    },
    "sklearn.svm.SVC": {
        "C": Real(1e-6, 20.0),
        "gamma": Real(1e-6, 10.0),
        "tol": Real(1e-6, 10.0),
        "class_weight": HDReal((1.0e-5, 1.0e-5), (20.0, 20.0)),
    },
    "lightgbm.LGBMClassifier": {
        "boosting_type": Categorical(["gbdt", "dart", "goss", "rf"]),
        "num_leaves": Integer(2, 100),
        "learning_rate": Real(1.0e-7, 1.0 - 1.0e-6),  # prior='uniform'),
        "n_estimators": Integer(5, 250),
        "min_split_gain": Real(0.0, 1.0),  # prior='uniform'),
        "subsample": Real(1.0e-6, 1.0),  # prior='uniform'),
        "importance_type": Categorical(["split", "gain"]),
    },
    # Preprocesssors
    "sklearn.preprocessing.StandardScaler": {
        "with_mean": Categorical([True, False]),
        "with_std": Categorical([True, False]),
    },
    "skrebate.ReliefF": {
        "n_features_to_select": Integer(2, 10),
        "n_neighbors": Integer(2, 10),
    },
    # Sensitivity Analysis
    "SKSurrogate.sensapprx.SensAprx": {
        "n_features_to_select": Integer(2, 20),
        "method": Categorical(["sobol", "morris", "delta-mmnt"]),
        "regressor": Categorical([None, regressor]),
    },
}
import warnings

warnings.filterwarnings("ignore", category=Warning)


genetic_data = pd.read_csv(
    "https://github.com/EpistasisLab/scikit-rebate/raw/master/data/"
    "GAMETES_Epistasis_2-Way_20atts_0.4H_EDM-1_1.tsv.gz",
    sep="\t",
    compression="gzip",
)
X, y = genetic_data.drop("class", axis=1).values, genetic_data["class"].values

A = AML(config=config, length=3, check_point="./", verbose=2)
A.eoa_fit(X, y, max_generation=10, num_parents=10)
print(A.get_top(5))
```

In order to perform an exhaustive search on all possible pipelines just replace the last line with the following:

```
A.fit(X, y)
```

We can retrieve the top *n* models via `A.get_top(n)`.

Example 2: The following is a regression based on Airfoil Self-Noise Data Set data:

```
# Find an optimum regression pipeline

import pandas as pd
import numpy as np
from sklearn.model_selection import RandomizedSearchCV
from sklearn.kernel_ridge import KernelRidge
from sklearn.gaussian_process.kernels import Matern, Sum, ExpSineSquared
from SKSurrogate import *

config = {
    # Regressors
    "sklearn.linear_model.LinearRegression": {"normalize": Categorical([True, False])}
↪,
    "sklearn.kernel_ridge.KernelRidge": {
        "alpha": Real(1.0e-4, 10.0),
        "kernel": Categorical(
            [
                Sum(Matern(), ExpSineSquared(l, p))
                for l in np.logspace(-2, 2, 10)
                for p in np.logspace(0, 2, 10)
            ]
        ),
    },
    # Preprocesssors
    "sklearn.preprocessing.StandardScaler": {
        "with_mean": Categorical([True, False]),
        "with_std": Categorical([True, False]),
    },
    "sklearn.preprocessing.Normalizer": {"norm": Categorical(["l1", "l2", "max"])},
    # Feature Selectors
    "sklearn.feature_selection.VarianceThreshold": {"threshold": Real(0.0, 0.3)},
}
import warnings

warnings.filterwarnings("ignore", category=Warning)

df = pd.read_csv(
    "https://archive.ics.uci.edu/ml/machine-learning-databases/00291/airfoil_self_
↪noise.dat",
    sep="\t",
    names=["Frequency", "Angle", "length", "velocity", "thickness", "level"],
)
X = df.drop("level", axis=1).values
y = df["level"].values

A = AML(
    config=config,
    length=3,
    check_point="./",
    verbose=2,
    scoring="neg_mean_squared_error",
)
A.eoa_fit(X, y, max_generation=12, num_parents=12)
print(A.get_top(5))
```

8.1 Some Technical Notes

It should be evident from the example that the *config* dictionary's keys could point to any module that is available from the working folder. The only constraint is that the classes being used must inherit from `sklearn.base.BaseEstimator`, `RegressorMixin`, `ClassifierMixin`, `TransformerMixin` or `imblearn.base.SamplerMixin`, `BaseSampler`.

The last estimator will always be selected from either `RegressorMixin` or `ClassifierMixin`. The case of `imblearn.base.SamplerMixin`, `BaseSampler` can only occur at the beginning of the pipeline. The rest could be `RegressorMixin`, `ClassifierMixin` or `TransformerMixin`.

8.1.1 Stacking

If a non `TransformerMixin` occurs in the middle, then by `StackingEstimator` it will transform the data to append columns based on the outcome of `RegressorMixin` or `ClassifierMixin`.

8.1.2 Permutation Importance

If `sklearn.pipeline.FeatureUnion` is included within the config dictionary, in the scope of a pipeline two scenarios are plausible:

- **'FeatureUnion' is followed by a series of transformations: in this case *FeatureUnion*** does exactly what is expected, i.e., gathers all the feature outputs of transformers;
- **'FeatureUnion' is followed by a mixture of transformations and estimators: then** *SKSurrogate* uses `eli5.sklearn.PermutationImportance` to weight the features based on the estimators and `AML`'s scoring and then selects top features via `sklearn.feature_selection.SelectFromModel`.

Not all transformers select a subset of of features (e.g., *Normalizer* or *StandardScaler*). If *FeatureUnion* is followed by such transformers, it does not have any effect on the outcome of the transformer. If the transformer selects a subset of features (*VarianceThreshold*, *skrebate.ReliefF*) then *FeatureUnion* collects the outcomes and returns the union. This is also true for *PermutationImportance*. The *FeatureUnion* affects the following transformers and estimators until it reaches the last step or a transformer which is not a feature selector. Subclasses of `sklearn.feature_selection.base.SelectorMixin` are considered as feature selectors. Also, the following transformers are considered as feature selectors:

- *FactorAnalysis*
- *FastICA*
- *IncrementalPCA*
- *KernelPCA*
- *LatentDirichletAllocation*
- *MiniBatchDictionaryLearning*
- *MiniBatchSparsePCA*
- *NMF*
- *PCA*
- *SparsePCA*
- *TruncatedSVD*
- *VarianceThreshold*

- *LocallyLinearEmbedding*
- *Isomap*
- *MDS*
- *SpectralEmbedding*
- *TSNE*
- *sksurrogate.SensAprx*
- *skrebate.ReliefF*
- *skrebate.SURF*
- *skrebate.SURFstar*
- *skrebate.MultiSURF*
- *skrebate.MultiSURFstar*
- *skrebate.TuRF*

8.1.3 imblearn pipelines

If an `imblearn` sampler is included in the *config* dictionary, then `imblearn.pipeline.Pipeline` will be used instead of `sklearn.pipeline.Pipeline` which enables the Pipeline to use imblearn samples too.

8.1.4 Categorical Variables

In case there are fields in the data that need to be treated as categorical, one could provide a list of indices through *cat_cols*. Then, the data will be transformed via `category_encoders.one_hot.OneHotEncoder` before being passed to the pipelines.

CHAPTER 9

Automated Data Type Recognition

In most ML scenarios, most of the development and deployment tasks deal with data input pipelines. A data pipeline handles data intake, linkage, type detection, and missing data imputation. SKSurrogate covers the later two stapes automatically and allows for customization as well. This is done via the `DataProcess` module.

Currently, the DataProcess module identifies the following data types automatically:

- Binary
- Categorical
- Date/Time
- Float
- Integer
- Label
- Text
- Objects

We note that the Object type could include various types which may have a known structure but are not implemented in the module yet.

The following example demonstrates the basic functions of the DataProcess module.

9.1 Example:

Randomly generated dataframe with various types of data:

```
import numpy as np
import pandas as pd
import random
from lorem_text import lorem
```

(continues on next page)

(continued from previous page)

```
N = 100
categorical = np.array([random.choice(['Cat01', 'Cat02', 'Cat03', 'Cate04', None])␣
↪for _ in range(N)])
binary = np.array([random.choice(['Bin0', 'Bin1', None]) for _ in range(N)])
float1 = np.random.uniform(low=-2.0, high=4.0, size=N)
float2 = np.random.uniform(low=0.0, high=10.0, size=N)
int1 = np.random.randint(0, high=20, size=N)

def random_str(max_len=15):
    chars = [' '] + [chr(_) for _ in range(ord('a'), ord('z')+1)] + [' ']+  [chr(_)␣
↪for _ in range(ord('A'), ord('Z')+1)] + [' ']
    ln = random.randint(0, max_len)
    rand_list = [random.choice(chars) for _ in range(ln)]
    return ''.join(rand_list)

def random_date(init_date, date_range=30):
    offset = random.randint(0, date_range)
    new_date = np.datetime64(init_date) + offset
    return new_date

strs = [random_str() for _ in range(N)]
dates = [random_date("2021-03-01", 60) for _ in range(N)]

texts = [lorem.sentence() for _ in range(N)]

frame = dict()
frame['categorical'] = categorical
frame['binary'] = binary
frame['float1'] = float1
frame['float2'] = float2
frame['int1'] = int1
frame['str'] = strs
frame['date'] = dates
frame['txt'] = texts
df = pd.DataFrame(frame)
df = df.astype({'txt':pd.StringDtype()})
```

Import and process the sample dataframe:

```
from SKSurrogate import *
A = DataPreprocess(df)
A.deduce_types()
A.deduced_types
```

which returns:

```
{'float64': ['float1', 'float2'],
 'int64': ['int1'],
 'datetime64': ['date'],
 'other': [],
 'text': ['txt'],
 'binary': ['binary'],
 'categorical': ['categorical'],
 'label': ['str'],
 'obsolete': []}
```

Then:

```
A.encode()
print(A.steps)
```

The output is a SK-Learn compatible pipeline:

```
[('OneHot',
  OneHotEncoder(cols=['categorical'], drop_invariant=True,
                handle_missing='return_nan', handle_unknown='return_nan')),
 ('Ordinal',
  OrdinalEncoder(cols=['binary', 'str'], handle_missing='return_nan',
                 handle_unknown='return_nan',
                 mapping=[{'col': 'binary', 'mapping': {'Bin0': 0, 'Bin1': 1}},
                          {'col': 'str',
                           'mapping': {'': 0, ' ': 1, ' DmTHDRQErIhF': 2,
                                       ' FmseqO': 3, ' j knr': 4, ' pcG': 5,
                                       'AVJVq nsqyHRpM': 6, 'AYihJxhUbN ': 7,
                                       'Agpg': 8, 'C': 9, 'CKJ': 10, 'CcnGK': 11,
                                       'D': 12, 'DkMstNYdjoRj ': 13, 'EITp': 14,
                                       'FAWrCVv': 15, 'FKgFwuGLmQqLR': 16,
                                       'FVtvoWBCEEi': 17, 'G T oVPh': 18,
                                       'GAxyFGqpzrJXe': 19, 'GYfNntcQww': 20,
                                       'HB euaV YFIb': 21, 'IENmSCFiAECp': 22,
                                       'IGZRolGBCKLsyg': 23,
                                       'J mNPFImkjd iRw': 24, 'JW': 25,
                                       'KSKpIlRm': 26, 'KevYeZyrsvwY': 27,
                                       'KhNjalpZkqxFGBC': 28,
                                       'KjtCfjg PZrx k ': 29, ...}}])),
 ('Date2Num', DateTime2Num(cols=['date'])),
 ('Impute', IterativeImputer())]
```

CHAPTER 10

A machine learning progress tracker

When exploring for reasonable methods to model a problem, usually the search quickly results in a large number of candidates and it becomes very difficult to keep track of all models and their performance measures. There are various existing solutions, usually forcing to follow a particular workflow, which could be very beneficial. The present module tends to be very light, with minimal dependency and minimal effect of the workflow.

mltrace loves the scikit-learn compatible models and provides many tools to work with them. The vision and hence design of *mltrace* is based on the believe that most machine learning projects can be turned into a study on a certain dataset. Although, as the project progresses, features may be added to or dropped from the original data, but in most cases there is a systematic method to derive these changes from the original source. Therefore, *mltrace* isolates a task together with a dataset and the associated models.

Let us make up a sample classification task and trace the models via *mltrace*.

Step 0. Make a sample classification dataset:

```
# import requires libraries
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from SKSurrogate import *
# make up a classification dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=6, n_
↪redundant=2)
Xy = np.hstack((X, np.reshape(y, (-1, 1))))
# make up some names for columns of the data
cols = ['cl%d'%(_+1) for _ in range(10)] + ['target']
# turn it into a pandas DataFrame
df = np2df(Xy, cols)()
```

Step 1. Initiate the tracker and register the data:

```
# initialize the tracker with a task called 'sample'
MLTr = mltrack('sample', db_name="sample.db")
# register the data
MLTr.RegisterData(df, 'target')
```

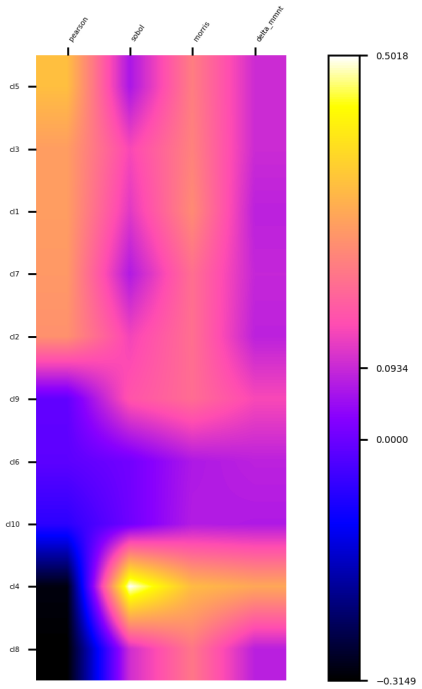
(continues on next page)

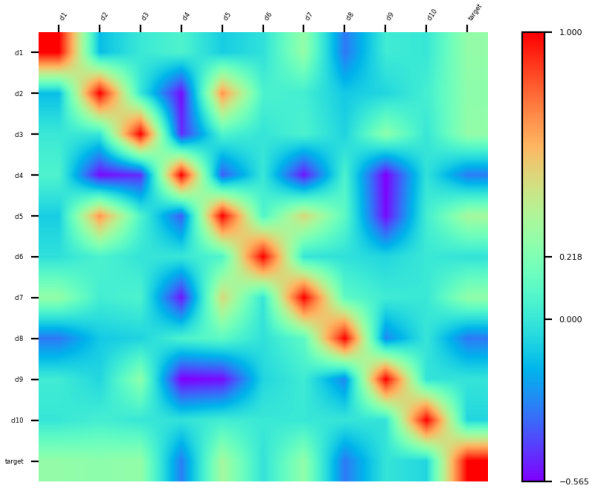
(continued from previous page)

```
# modify the description of the task
MLTr.UpdateTask({'description': "This is a sample task to demonstrate\\
        capabilities of the mltrace."})
```

Step 2. Get to know the data by visualizing correlations and sensitivities:

```
from sklearn.gaussian_process.kernels import Matern, Sum, ExpSineSquared
from sklearn.kernel_ridge import KernelRidge
from sklearn.model_selection import RandomizedSearchCV
# use a regressor to approximate the data
param_grid_kr = {"alpha": np.logspace(-4, 1, 20),
                 "kernel": [Sum(Matern(), ExpSineSquared(l, p))
                            for l in np.logspace(-2, 2, 10)
                            for p in np.logspace(0, 2, 10)]}
rgs = RandomizedSearchCV(KernelRidge(),
        param_distributions=param_grid_kr, n_iter=10, cv=2)
# ask for specific weights to be calculated and recorded
MLTr.FeatureWeights(regressor=rgs,
        weights=('pearson', 'sobol', 'morris', 'delta-mmnt'))
# visualise
plt1 = MLTr.heatmap(sort_by='pearson')
plt1.show()
cor = df.corr()
plt2 = p = MLTr.heatmap(cor, idx_col=None, cmap='rainbow')
plt2.show()
```





Step 3. Examine and log a random forest model and its metrics:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score, ShuffleSplit
# retrieve data
```

(continues on next page)

(continued from previous page)

```
X, y = MLTr.get_data()
# init classifier
clf = RandomForestClassifier(n_estimators=50)
# log the classifier
clf = MLTr.LogModel(clf, "RandomForestClassifier(50)")
# find the average metrics
print(MLTr.LogMetrics(clf, cv=ShuffleSplit(5, .25)))
# {'accuracy': 0.8816, 'auc': 0.9504791437613562, 'precision': 0.9029848807772192,
# 'f1': 0.8782525829634803, 'recall': 0.8554950898148654, 'mcc': 0.7640884799286114,
# 'logloss': 4.089427586800243, 'variance': None, 'max_error': None, 'mse': None,
# 'mae': None, 'r2': None}
```

Step 4. Search for best (in terms of accuracy) classifier as a combination of `naive_bayes.GaussianNB`, `linear_model.LogisticRegression`, `lightgbm.LGBMClassifier`, `preprocessing.StandardScaler`, and `preprocessing.Normalizer`:

```
from SKSurrogate import *
# set up the confic dictionary
config = {
    # estimators
    'sklearn.naive_bayes.GaussianNB': {
        'var_smoothing': Real(1.e-9, 2.e-1)
    },
    'sklearn.linear_model.LogisticRegression': {
        'penalty': Categorical(["l1", "l2"]),
        'C': Real(1.e-6, 10.),
        "class_weight": HDReal((1.e-5, 1.e-5), (20., 20.))
    },
    "lightgbm.LGBMClassifier": {
        "boosting_type": Categorical(['gbdt', 'dart', 'goss', 'rf']),
        "num_leaves": Integer(2, 100),
        "learning_rate": Real(1.e-7, 1. - 1.e-6),  # prior='uniform'),
        "n_estimators": Integer(5, 250),
        "min_split_gain": Real(0., 1.),  # prior='uniform'),
        "subsample": Real(1.e-6, 1.),  # prior='uniform'),
        "importance_type": Categorical(['split', 'gain'])
    },
    # preprocessing
    'sklearn.preprocessing.StandardScaler': {
        'with_mean': Categorical([True, False]),
        'with_std': Categorical([True, False]),
    },
    'sklearn.preprocessing.Normalizer': {
        'norm': Categorical(['l1', 'l2', 'max'])
    },
}
# initiate and perform the search
A = AML(config=config, length=3, check_point='./sample/', verbose=1)
A.eoa_fit(X, y, max_generation=15, num_parents=20)
# retrieve and log the best
eoa_clf = A.best_estimator_
eoa_clf = MLTr.LogModel(eoa_clf, "Best of EOA Surrogate Search")
print(MLTr.LogMetrics(eoa_clf, cv=ShuffleSplit(5, .25)))
MLTr.PreserveModel(eoa_clf)
# {'accuracy': 0.8824, 'auc': 0.9207884992789751, 'precision': 0.8930688738450767,
# 'f1': 0.8789651291713657, 'recall': 0.8664344690110679, 'mcc': 0.7657250436230718,
```

(continues on next page)

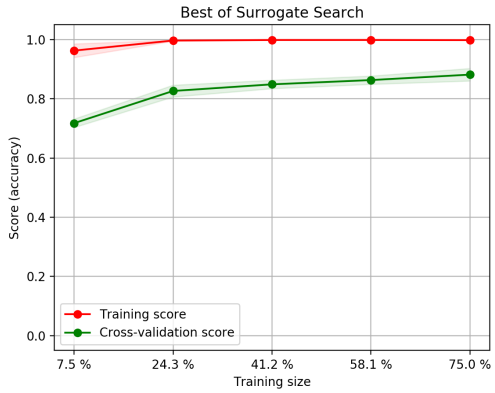
(continued from previous page)

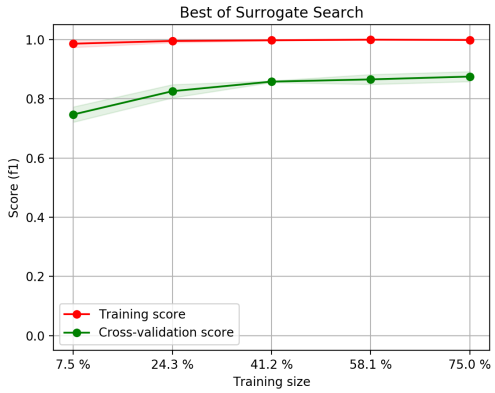
```
# 'logloss': 4.061801043429923, 'variance': None, 'max_error': None, 'mse': None,
# 'mae': None, 'r2': None}
```

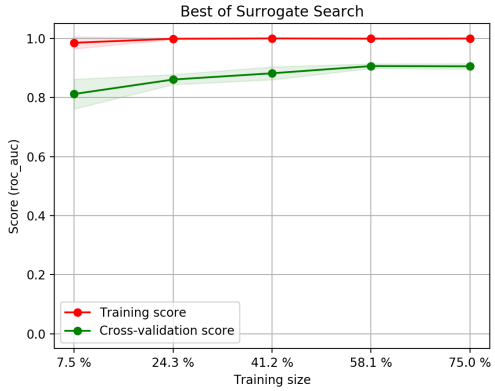
Step 5. Plot learning curves for accuracy, F_1, area under ROC, calibration lift and cumulative curves for the two models:

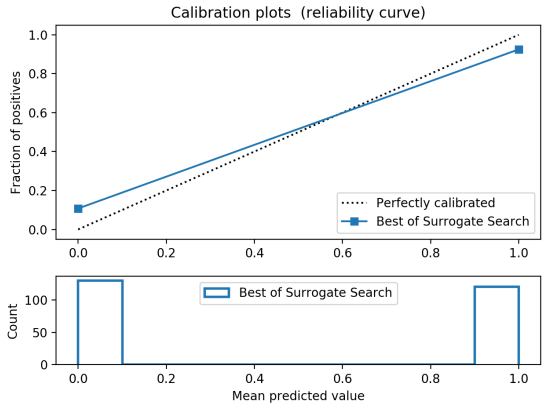
```
# the best of EOA Surrogate Search
MLTr.plot_learning_curve(eoa_clf, "Best of Surrogate Search", cv=ShuffleSplit(5, .
↪25), measure='accuracy')
MLTr.plot_learning_curve(eoa_clf, "Best of Surrogate Search", cv=ShuffleSplit(5, .
↪25), measure='f1')
MLTr.plot_learning_curve(eoa_clf, "Best of Surrogate Search", cv=ShuffleSplit(5, .
↪25), measure='roc_auc')
MLTr.plot_calibration_curve(eoa_clf, "Best of Surrogate Search")
MLTr.plot_cumulative_gain(eoa_clf, title="Best: Cumulative Gains Curve")
MLTr.plot_lift_curve(eoa_clf, title="Best: Lift Curve")
# Random Forest
MLTr.plot_learning_curve(clf, "Random Forest", cv=ShuffleSplit(5, .25), measure=
↪'accuracy')
MLTr.plot_learning_curve(clf, "Random Forest", cv=ShuffleSplit(5, .25), measure=
↪'f1')
MLTr.plot_learning_curve(clf, "Random Forest", cv=ShuffleSplit(5, .25), measure=
↪'roc_auc')
MLTr.plot_calibration_curve(clf, "Random Forest")
MLTr.plot_cumulative_gain(clf, title="Random Forest: Cumulative Gains Curve")
MLTr.plot_lift_curve(clf, title="Random Forest: Lift Curve")
```

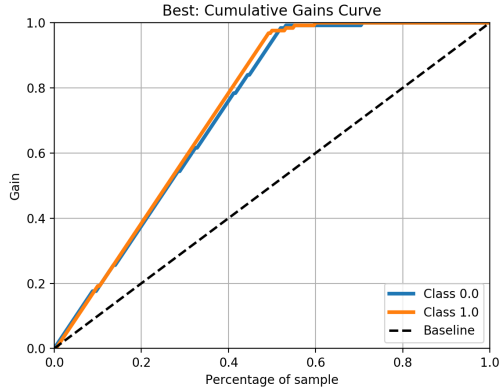
The Best of EOA

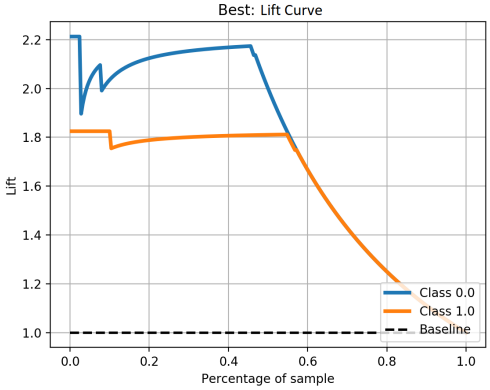




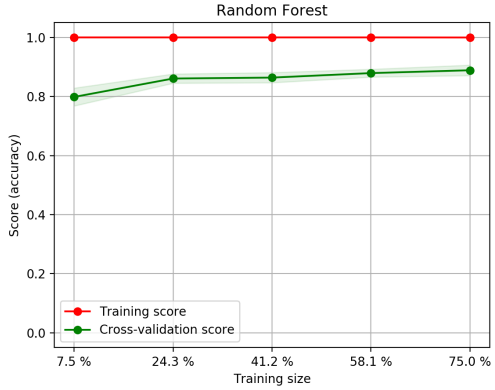


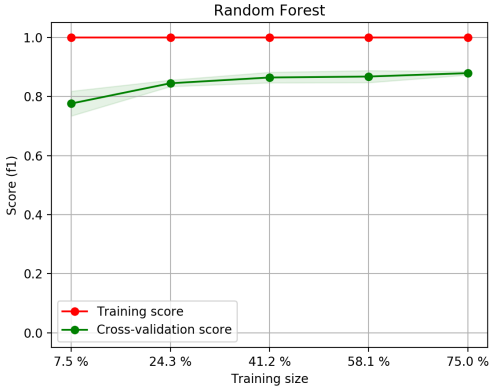


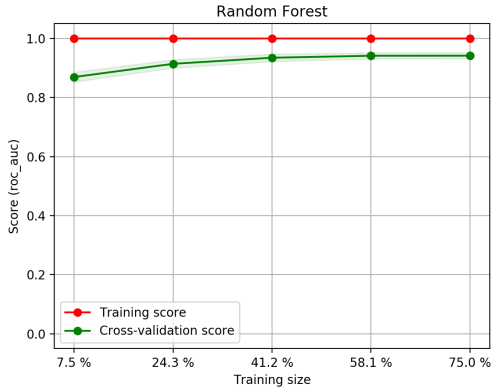


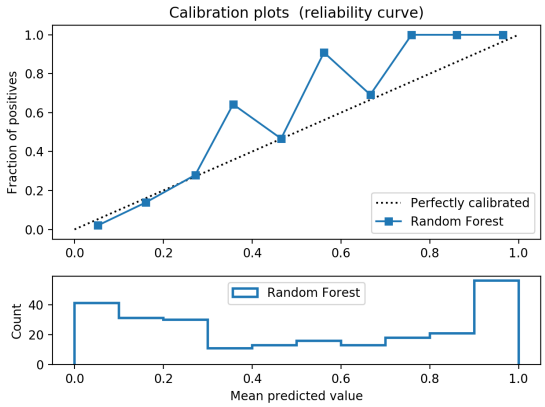


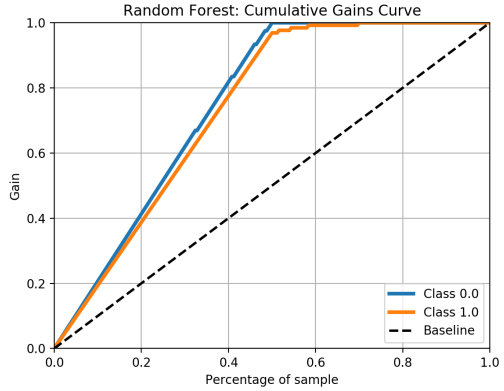
Random Forest

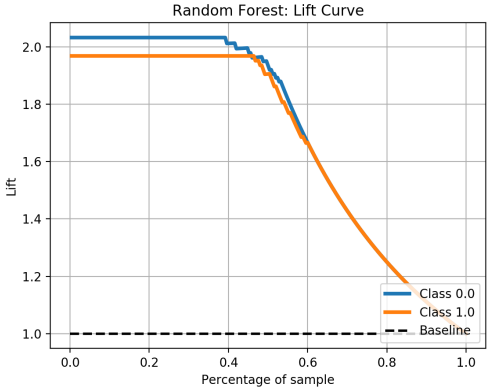












10.1 SVM Classifier with RBF v.s. SGDClassifier with Kernels

We try to see whether SVM classifiers with RBF kernel and SGDClassifres with kernel can compare. We set up a quick `SKSurrogate` search with `SVC` and *NuSVC* as classifiers and another quick `SKSurrogate` search over `SGDClassifier`, `Nystroem` and `RBFSampler` kernels:

```
import numpy as np
from SKSurrogate import *
import warnings
warnings.filterwarnings("ignore", category=Warning)

from sklearn.model_selection import RandomizedSearchCV
from sklearn.kernel_ridge import KernelRidge
from sklearn.gaussian_process.kernels import Matern, Sum, ExpSineSquared
param_grid_krr = {"alpha": np.logspace(-4, 0, 5),
                  "kernel": [Sum(Matern(), ExpSineSquared(l, p))
                             for l in np.logspace(-2, 2, 10)
                             for p in np.logspace(0, 2, 10)]}
regressor = RandomizedSearchCV(KernelRidge(), param_distributions=param_grid_krr, n_
↪iter=7, cv=2)

config = {
    # estimators
    'sklearn.svm.SVC': {
        "C": Real(1e-6, 20.),
        "gamma": Real(1e-6, 10.),
```

(continues on next page)

(continued from previous page)

```
        "tol": Real(1e-6, 10.),
        "class_weight": HDReal((1.e-5, 1.e-5), (20., 20.))
    },
    'sklearn.svm.NuSVC': {
        'nu': Real(1e-5, 1.),
        "gamma": Real(1e-6, 10.),
        "tol": Real(1e-6, 10.),
        "class_weight": HDReal((1.e-5, 1.e-5), (20., 20.))
    },
    # preprocessing
    'sklearn.preprocessing.StandardScaler': {
        'with_mean': Categorical([True, False]),
        'with_std': Categorical([True, False]),
    },
    'sklearn.feature_selection.VarianceThreshold': {
        'threshold': Real(0., .3)
    },
    'sklearn.preprocessing.Normalizer': {
        'norm': Categorical(['l1', 'l2', 'max'])
    },
}

MLTr = mltrack('sample', db_name="sample.db")
X, y = MLTr.get_data()


A_svm = AML(config=config, length=3, check_point='./svm/', verbose=1)
A_svm.eoa_fit(X, y, max_generation=10, num_parents=10)
print(A_svm.get_top(4))
```

Which results in:

```
OrderedDict([(('sklearn.feature_selection.VarianceThreshold',
           'sklearn.preprocessing.Normalizer',
           'sklearn.svm.SVC'),
          (Pipeline(memory=None,
                steps=[('stp_0', VarianceThreshold(threshold=0.20066923736000097)), (
↪'stp_1', Normalizer(copy=True, norm='l2')), ('stp_2', SVC(C=15.110221076172207,␣
↪cache_size=200,
              class_weight={0.0: 13.581338880577112, 1.0: 3.1546898782179706},
              coef0=0.0, decision_function_shape='ovr', degree=3, gamma=10.0,
              kernel='rbf', max_iter=-1, probability=False, random_state=None,
              shrinking=True, tol=1e-06, verbose=False))]),
           -0.9253333333333333)),
         (('sklearn.preprocessing.StandardScaler',
           'sklearn.preprocessing.Normalizer',
           'sklearn.svm.SVC'),
          (Pipeline(memory=None,
                steps=[('stp_0', StandardScaler(copy=True, with_mean=True, with_
↪std=False)), ('stp_1', Normalizer(copy=True, norm='l1')), ('stp_2', SVC(C=14.
↪396152757785778, cache_size=200,
              class_weight={0.0: 11.644799650485178, 1.0: 14.834346896165036},
              coef0=0.0, decision_function_shape='ovr', degree=3,
              gamma=1.387860853481898, kernel='rbf', max_iter=-1, probability=False,
              random_state=None, shrinking=True, tol=1.3386685658572253,␣
↪verbose=False))]),
           -0.9253333333333333)),
```

(continues on next page)

(continued from previous page)

```
         (('sklearn.svm.NuSVC',
           'sklearn.preprocessing.Normalizer',
           'sklearn.svm.SVC'),
          (Pipeline(memory=None,
                steps=[('stp_0', StackingEstimator(decision=True,
                    estimator=NuSVC(cache_size=200,
               class_weight={0.0: 16.514247793012903, 1.0: 19.743755570932407},
               coef0=0.0, decision_function_shape='ovr', degree=3,
               gamma=2.879810799993445, kernel='rbf', max_iter=-1,
               nu=5.366323874825201e-05, pr...robability=False,
              random_state=None, shrinking=True, tol=0.0002612195258127529,
              verbose=False))]), -0.9226666666666666)),
         (('sklearn.feature_selection.VarianceThreshold',
           'sklearn.preprocessing.Normalizer',
           'sklearn.svm.NuSVC'),
          (Pipeline(memory=None,
                steps=[('stp_0', VarianceThreshold(threshold=0.10372588511430014)), (
↪'stp_1', Normalizer(copy=True, norm='l2')), ('stp_2', NuSVC(cache_size=200,
               class_weight={0.0: 4.756565216129669, 1.0: 14.36176825433476},
               coef0=0.0, decision_function_shape='ovr', degree=3,
               gamma=5.437271690133034, kernel='rbf', max_iter=-1,
               nu=0.6557563687772239, probability=False, random_state=None,
               shrinking=True, tol=2.1918030657741365, verbose=False))]),
           -0.9186666666666666)))])
```

And for `SGDClassifier`:

```
import numpy as np
from SKSurrogate import *
import warnings
warnings.filterwarnings("ignore", category=Warning)

from sklearn.model_selection import RandomizedSearchCV
from sklearn.kernel_ridge import KernelRidge
from sklearn.gaussian_process.kernels import Matern, Sum, ExpSineSquared
param_grid_krr = {"alpha": np.logspace(-4, 0, 5),
                  "kernel": [Sum(Matern(), ExpSineSquared(l, p))
                             for l in np.logspace(-2, 2, 10)
                             for p in np.logspace(0, 2, 10)]}
regressor = RandomizedSearchCV(KernelRidge(), param_distributions=param_grid_krr, n_
↪iter=7, cv=2)

config = {
    # estimators
    'sklearn.linear_model.SGDClassifier': {
        'loss': Categorical(['hinge', 'log', 'modified_huber', 'squared_hinge',
↪'perceptron']),
        'penalty': Categorical(['none', 'l2', 'l1', 'elasticnet']),
        'alpha': Real(1.e-5, .9999),
        'l1_ratio': Real(0., 1.),
        'tol': Real(1.e-5, 1.),
        'class_weight': HDReal((1.e-5, 1.e-5), (20., 20.))
    },
    # preprocessing
    'sklearn.preprocessing.StandardScaler': {
        'with_mean': Categorical([True, False]),
        'with_std': Categorical([True, False]),
```

(continues on next page)

(continued from previous page)

```
    },
    'sklearn.feature_selection.VarianceThreshold': {
        'threshold': Real(0., .3)
    },
    'sklearn.preprocessing.Normalizer': {
        'norm': Categorical(['l1', 'l2', 'max'])
    },
    # Transformers
    'sklearn.kernel_approximation.Nystroem': {
        'kernel': Categorical(['rbf', 'poly', 'sigmoid']),
        'gamma': Real(1.e-6, 10.),
        'n_components': Integer(10, 120)
    },
    'sklearn.kernel_approximation.RBFSampler': {
        'gamma': Real(1.e-6, 10.),
        'n_components': Integer(10, 120)
    },
}

MLTr = mltrack('sample', db_name="sample.db")
X, y = MLTr.get_data()


A_sgd = AML(config=config, length=3, check_point='./svm/', verbose=2)#, cat_cols=[5])
A_sgd.fit(X, y)
print(A_sgd.get_top(4))
```

Which results in:

```
OrderedDict([(('sklearn.linear_model.SGDClassifier',
                'sklearn.kernel_approximation.Nystroem',
                'sklearn.linear_model.SGDClassifier'),
               (Pipeline(memory=None,
                     steps=[('stp_0', StackingEstimator(decision=True,
                         estimator=SGDClassifier(alpha=1.0186882143892309e-05,␣
↪average=False,
                       class_weight={0.0: 1e-05, 1.0: 1e-05}, early_stopping=False,
                       epsilon=0.1, eta0=0.0, fit_intercept=True,
                       l1_ratio=0.16668760571866542, learning_rate='op...om_state=None,
↪ shuffle=True,
                       tol=1.0, validation_fraction=0.1, verbose=0, warm_
↪start=False))])],
                -0.8946666666666667)),
              (('sklearn.preprocessing.Normalizer',
                'sklearn.kernel_approximation.RBFSampler',
                'sklearn.linear_model.SGDClassifier'),
               (Pipeline(memory=None,
                     steps=[('stp_0', Normalizer(copy=True, norm='l1')), ('stp_1',␣
↪RBFSampler(gamma=5.09346829872262, n_components=120, random_state=None)), ('stp_2',␣
↪SGDClassifier(alpha=1e-05, average=False,
                       class_weight={0.0: 6.299501235657723, 1.0: 14.399482243823948},
                       early_stopping=False, epsilon=0.1,..._state=None, shuffle=True,
                       tol=1e-05, validation_fraction=0.1, verbose=0, warm_
↪start=False))])],
                -0.8946666666666667)),
              (('sklearn.kernel_approximation.Nystroem',
                'sklearn.preprocessing.Normalizer',
```

(continues on next page)

(continued from previous page)

```
               'sklearn.linear_model.SGDClassifier'),
              (Pipeline(memory=None,
                        steps=[('stp_0', Nystroem(coef0=None, degree=None, gamma=7.
↪102137254366565, kernel='poly',
                        kernel_params=None, n_components=51, random_state=None)), ('stp_1
↪', Normalizer(copy=True, norm='l2')), ('stp_2', SGDClassifier(alpha=0.
↪4030001737762923, average=False,
                          class_weight={0.0: 16.5900487...shuffle=True, tol=0.
↪6700383879862661,
                          validation_fraction=0.1, verbose=0, warm_start=False))]),
               -0.8906666666666667)),
             (('sklearn.preprocessing.Normalizer',
               'sklearn.kernel_approximation.Nystroem',
               'sklearn.linear_model.SGDClassifier'),
              (Pipeline(memory=None,
                        steps=[('stp_0', Normalizer(copy=True, norm='l2')), ('stp_1',␣
↪Nystroem(coef0=None, degree=None, gamma=5.928661960771689, kernel='poly',
                        kernel_params=None, n_components=117, random_state=None)), ('stp_2
↪', SGDClassifier(alpha=0.5112964074641659, average=False,
                          class_weight={0.0: 3.2816638...shuffle=True, tol=0.
↪7276228102426916,
                          validation_fraction=0.1, verbose=0, warm_start=False))]),
               -0.884))])
```

This shows a 3.06% loss in accuracy

CHAPTER 11

Code Documentation

11.1 Surrogate Random Search

This module provides basic functionality to optimize an expensive black-box function based on *Surrogate Random Search*. The Structured Random Search (SRS) method attempts to approximate an optimal solution to the following

minimize $f(x)$

subject to $g_i(x) \geq 0 \quad i = 1, \ldots, m,$

where arbitrary evaluations of f is not a viable option. The original random search itself is guarantee to converge to a local solution, but the convergence is usually very slow and most information about f is dismissed except for the best candidate. SRS tries to use all information acquired about f so far during the iterations. At i^{th} round of iteration SRS replaces f by a surrogate $\hat{f}_i$ that enjoys many nice analytical properties which make its optimization an easier task to overcome. Then by solving the above optimization problem with f replaced by $\hat{f}$ one gets a more informed candidate x_i for the next iteration. If a certain number of iterations do not result in a better candidate, the method returns to random sampling to collect more information about f. The surrogate function $\hat{f}_i$ can be found in many ways such as (non)linear regression, Gaussian process regression, etc. and *SurrogateSearch* do not have a preference. But by default it uses a polynomial regression of degree 3 if no regressor is provided. Any regressor following the architecture of *scikit-learn* is acceptable. Note that regressors usually require a minimum number of data points to function properly.

There are various ways for sampling a random point in feasible space which affects the performance of SRS. *SurrogateSearch* implements two methods: *BoxSample* and *SphereSample*. One can choose whether to shrink the volume of the box or sphere that the sample is selected from too.

class structsearch.**BaseSample**(***kwargs*)

This is the base class for various sampling methods.

Parameters

- **init_radius** – **optional** (default=2.); positive real number indicating the initial radius of the local search ball.
- **contraction** – **optional** (default=.0); the contraction factor which must be a positive real less than 1.
- **ineq** – **optional**; a list of functions whose positivity region will be the acceptable condition.

- **bounds** – **optional**; the list of (ordered) tuples determining the bound of each component.

check_constraints(*point*)
Checks constraints on the sample if provided

Parameters **point** – the candidate to be checked

Returns *boolean* True or False for if all constraints hold or not.

class structsearch.**BoxSample**(***kwargs*)
Generates samples out of a box around a given center.

Parameters

- **init_radius** – *float* the initial half-length of the edges of the sampling box; default: 2.
- **contraction** – *float* the contraction factor for repeated sampling.

sample(*centre*, *cntrctn=1.0*)
Samples a point out of a box centered at *centre*

Parameters

- **centre** – *numpy.array* the center of the box
- **cntrctn** – *float* customized contraction factor

Returns *numpy.array* a new sample

class structsearch.**Categorical**(*items*, ***kwargs*)
A list of possible values fr the search algorithm to choose from.

Parameters **items** – A list of possible values for a parameter

class structsearch.**CompactSample**(***kwargs*)
Generates samples uniformly out of a box.

sample(*centre*, *cntrctn=1.0*)
Samples a point out of a box centered at *centre*

Parameters

- **centre** – *numpy.array* the center of the box
- **cntrctn** – *float* customized contraction factor

Returns *numpy.array* a new sample

class structsearch.**HDReal**(*a*, *b*, ***kwargs*)
An *n* dimensional box of real numbers corresponding to the classification groups (e.g. *class_weight*). *a* is the list of lower bounds and *b* is the list of upper bounds.

Parameters

- **a** – a tuple of lower bounds for each dimension
- **b** – a tuple of upper bounds for each dimension

class structsearch.**Integer**(*a=None*, *b=None*, ***kwargs*)
The range of possible values for an integer variable; *a* is the minimum and *b* is the maximum. Defaults are + and - infinity.

Parameters

- **a** – the lower bound for the integer interval defined by instance (accepting '-numpy.inf')
- **b** – the upper bound for the integer interval defined by instance (accepting 'numpy.inf')

class structsearch.**Real**(*a=None*, *b=None*, ***kwargs*)

The range of possible values for a real variable; *a* is the minimum and *b* is the maximum. Defaults are + and - infinity.

Parameters

- **a** – the lower bound for the (closed) interval defined by instance (accepting '-numpy.inf')
- **b** – the upper bound for the (closed) interval defined by instance (accepting 'numpy.inf')

class structsearch.**SphereSample**(***kwargs*)

Generates samples out of an sphere around a given center.

Parameters

- **init_radius** – *float* the initial radius of the sampling sphere; default: 2.
- **contraction** – *float* the contraction factor for repeated sampling.

sample(*centre*, *cntrctn=1.0*)

Samples a point out of an sphere centered at *centre*

Parameters

- **centre** – *numpy.array* the center of the sphere
- **cntrctn** – *float* customized contraction factor

Returns *numpy.array* a new sample

class structsearch.**SurrogateRandomCV**(*estimator*, *params*, *scoring=None*, *fit_params=None*, *n_jobs=-1*, *refit=True*, *cv=None*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *error_score='raise'*, *return_train_score=True*, *max_iter=50*, *min_evals=25*, *regressor=None*, *sampling=<class 'structsearch.CompactSample'>*, *radius=None*, *contraction=0.95*, *search_sphere=False*, *optimizer='scipy'*, *scipy_solver='SLSQP'*, *task_name='optim_task'*, *warm_start=True*, *Continue=False*, *max_itr_no_prog=10000*, *ineqs=()*, *init=None*)

Surrogate Random Search optimization over hyperparameters.

The parameters of the estimator used to apply these methods are optimized by cross-validated search over parameter settings.

In contrast to GridSearchCV, not all parameter values are tried out, but rather a fixed number of parameter settings is sampled from the specified distributions. The number of parameter settings that are tried is given by *max_iter*.

Parameters

- **estimator** – estimator object. An object of that type is instantiated for each search point. This object is assumed to implement the scikit-learn estimator api. Either estimator needs to provide a `score` function, or `scoring` must be passed.
- **params** – dict Dictionary with parameters names (string) as keys and domains as lists of parameter ranges to try. Domains are either lists of categorical (string) values or 2 element lists specifying a min and max for integer or float parameters
- **scoring** – string, callable or *None*, default=None A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`. If `None`, the `score` method of the estimator is used.

- **max_iter** – int, default=50 Number of parameter settings that are sampled. max_iter trades off runtime vs quality of the solution. Consider increasing `n_points` if you want to try more parameter settings in parallel.
- **min_evals** – int, default=25; Number of random evaluations before employing an approximation for the response surface.
- **n_jobs** – int, default=-1; number of processes to run in parallel
- **fit_params** – dict, optional; Parameters to pass to the fit method.
- **pre_dispatch** – int, or string, optional; Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:
  - None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
  - An int, giving the exact number of total jobs that are spawned
  - A string, giving an expression as a function of n_jobs, as in '2*n_jobs'
- **cv** – int, cross-validation generator or an iterable, optional Determines the cross-validation splitting strategy. Possible inputs for cv are:
  - None, to use the default 3-fold cross validation,
  - integer, to specify the number of folds in a *(Stratified)KFold*,
  - An object to be used as a cross-validation generator.
  - An iterable yielding train, test splits.

  For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, `StratifiedKFold` is used. In all other cases, `KFold` is used.
- **refit** – boolean, default=True Refit the best estimator with the entire dataset. If "False", it is impossible to make predictions using this RandomizedSearchCV instance after fitting.
- **verbose** – int, default=0 Prints internal information about the progress of each iteration.

fit (*X*, *y=None*, *groups=None*, ***fit_params*)
Run fit with all sets of parameters.

Parameters

- **X** – array-like, *shape = [n_samples, n_features]* Training vector, where n_samples is the number of samples and n_features is the number of features.
- **y** – array-like, *shape = [n_samples]* or *[n_samples, n_output]*, optional; Target relative to X for classification or regression; None for unsupervised learning.
- **groups** – array-like, with shape *(n_samples,)*, optional; Group labels for the samples used while splitting the dataset into train/test set.
- **fit_params** – dict of *string -> object*; Parameters passed to the fit method of the estimator

Returns *self*

class structsearch.**SurrogateSearch**(*objective*, ***kwargs*)
An implementation of the Surrogate Random Search (SRS).

Parameters

- **objective** – a *callable*, the function to be minimized
- **ineq** – a list of callables which represent the constraints (default: [])
- **task_name** – *str* a name to refer to the optimization task, store & restore previously acquired (default: 'optim_task')
- **bounds** – a list of tuples of real numbers representing the bounds on each variable; default: None
- **max_iter** – *int* the maximum number of iterations (default: 50)
- **radius** – *float* the initial radius of sampling region (default: 2.)
- **contraction** – *float* the rate of radius contraction (default: .9)
- **sampling** – the sampling method either *BoxSample* or *SphereSample* (default *SphereSample*)
- **search_sphere** – *boolean* whether to fit the surrogate function on a neighbourhood of current candidate or over all sampled points (default: False)
- **deg** – *int* degree of polynomial regressor if one chooses to fit polynomial surrogates (default: 3)
- **min_evals** – *int* minimum number of samples before fitting a surrogate (default will be calculated as if the surrogate is a polynomial of degree 3)
- **regressor** – a regressor (scikit-learn style) to find a surrogate
- **scipy_solver** – *str* the scipy solver ('COBYLA' or 'SLSQP') to solve the local optimization problem at each iteration (default: 'COBYLA')
- **max_itr_no_prog** – *int* maximum number of iterations with no progress (default: infinity)
- **Continue** – *boolean* continues the progress from where it has been interrupted (default: False)
- **warm_start** – *boolean* use data from the previous attempts, but starts from the first iteration (default: False)
- **verbose** – *boolean* whether to report the progress on commandline or not (default: False)

progress()

Generates *matplotlib* plots that represent distributions of each variable and the progress in minimization.

Returns objective's process plot, variables' distributions

11.2 Evolutionary Optimization Algorithm

class eoa.**EOA**(*population*, *fitness*, ***kwargs*)

This is a base class acting as an umbrella to perform an evolutionary optimization algorithm.

Parameters

- **population** – The whole possible population as a list
- **fitness** – The fitness evaluation. Accepts an OrderedDict of individuals with their corresponding fitness and updates their fitness
- **init_pop** – default='UniformRand'; The python class that initiates the initial population

- **recomb** – default='UniformCrossover'; The python class that defines how to combine parents to produce children
- **mutation** – default='Mutation'; The python class that performs mutation on offspring population
- **termination** – default='MaxGenTermination'; The python class that determines the termination criterion
- **elitism** – default='Elites'; The python class that decides how to handel elitism
- **num_parents** – The size of initial parents population
- **parents_porp** – default=0.1; The size of initial parents population given as a portion of whole population (only used if *num_parents* is not given)
- **elits_porp** – default=0.2; The porportion of offspring to be replaced by elite parents
- **mutation_prob** – The probability that a component will be mutated (default: 0.05)
- **kwargs** –

class eoa.**MaxGenTermination**(***kwargs*)
Termination condition: Whether the maximum number of generations has been reached or not

class eoa.**UniformCrossover**(***kwargs*)
Recombination procedure.

class eoa.**UniformRand**(***kwargs*)
Initial population initiation.

11.3 Hilbert Space based regression

exception NpyProximation.**Error**(**args*)
Generic errors that may occur in the course of a run.

class NpyProximation.**FunctionBasis**
This class generates two typical basis of functions: Polynomials and Trigonometric

static Fourier(*n*, *deg*, *l=1.0*)
Returns the Fourier basis of degree *deg* in *n* variables with period *l*

Parameters

- **n** – number of variables
- **deg** – the maximum degree of trigonometric combinations in the basis
- **l** – the period

Returns the raw basis consists of trigonometric functions of degrees up to *n*

static Poly(*n*, *deg*)
Returns a basis consisting of polynomials in *n* variables of degree at most *deg*.

Parameters

- **n** – number of variables
- **deg** – highest degree of polynomials in the basis

Returns the raw basis consists of polynomials of degrees up to *n*

class NpyProximation.**FunctionSpace**(*dim=1*, *measure=None*, *basis=None*)

A class tha facilitates a few types of computations over function spaces of type $L_2(X, \mu)$

Parameters

- **dim** – the dimension of 'X' (default: 1)
- **measure** – an object of type *Measure* representing μ
- **basis** – a finite basis of functions to construct a subset of $L_2(X, \mu)$

FormBasis()

Call this method to generate the orthogonal basis corresponding to the given basis. The result will be stored in a property called `OrthBase` which is a list of function that are orthogonal to each other with respect to the measure `measure` over the given range `domain`.

Series(*f*)

Given a function *f*, this method finds and returns the coefficients of the series that approximates f as a linear combination of the elements of the orthogonal basis B. In symbols $\sum_{b \in B} \langle f, b \rangle b$.

Returns the list of coefficients $\langle f, b \rangle$ for $b \in B$

inner(*f*, *g*)

Computes the inner product of the two parameters with respect to the measure *measure*, i.e., $\int_X f \cdot g d\mu$.

Parameters

- **f** – callable
- **g** – callable

Returns the quantity of $\int_X f \cdot g d\mu$

project(*f*, *g*)

Finds the projection of f on g with respect to the inner product induced by the measure *measure*.

Parameters

- **f** – callable
- **g** – callable

Returns the quantity of $\frac{\langle f, g \rangle}{\|g\|_2} g$

class NpyProximation.**HilbertRegressor**(*deg=3*, *base=None*, *meas=None*, *fspace=None*)

Regression using Hilbert Space techniques Scikit-Learn style.

Parameters

- **deg** – int, default=3 The degree of polynomial regression. Only used if *base* is *None*
- **base** – list, default = None a list of function to form an orthogonal function basis
- **meas** – NpyProximation.Measure, default = None the measure to form the $L_2(\mu)$ space. If *None* a discrete measure will be constructed based on *fit* inputs
- **fspace** – NpyProximation.FunctionBasis, default = None the function subspace of $L_2(\mu)$, if *None* it will be initiated according to *self.meas*

fit(*X*, *y*)

Parameters

- **X** – Training data
- **y** – Target values

Returns *self*

predict(*X*)
Predict using the Hilbert regression method

Parameters **X** – Samples

Returns Returns predicted values

class NpyProximation.**Measure**(*density=None*, *domain=None*)
Constructs a measure μ based on *density* and *domain*.

Parameters

- **density** – the density over the domain: + if none is given, it assumes uniform distribution
  - if a callable *h* is given, then $d\mu = h(x)dx$
  - if a dictionary is given, then $\mu = \sum w_x \delta_x$ a discrete measure. The points x are the keys of the dictionary (tuples) and the weights w_x are the values.
- **domain** – if *density* is a dictionary, it will be set by its keys. If callable, then *domain* must be a list of tuples defining the domain's box. If None is given, it will be set to $[-1, 1]^n$

integral(*f*)
Calculates $\int_{domain} f d\mu$.

Parameters **f** – the integrand

Returns the value of the integral

norm(*p*, *f*)
Computes the norm-*p* of the *f* with respect to the current measure, i.e., $(\int_{domain} |f|^p d\mu)^{1/p}$.

Parameters

- **p** – a positive real number
- **f** – the function whose norm is desired.

Returns $\|f\|_{p,\mu}$

class NpyProximation.**Regression**(*points*, *dim=None*)
Given a set of points, i.e., a list of tuples of the equal lengths *P*, this class computes the best approximation of a function that fits the data, in the following sense:

- if no extra parameters is provided, meaning that an object is initiated like `R = Regression(P)` then calling `R.fit()` returns the linear regression that fits the data.
- if at initiation the parameter *deg=n* is set, then `R.fit()` returns the polynomial regression of degree *n*.
- if a basis of functions provided by means of an *OrthSystem* object (`R.SetOrthSys(orth)`) then calling `R.fit()` returns the best approximation that can be found using the basic functions of the *orth* object.

Parameters

- **points** – a list of points to be fitted or a callable to be approximated
- **dim** – dimension of the domain

SetFuncSpc(*sys*)
Sets the bases of the orthogonal basis

Parameters **sys** – *orthsys.OrthSystem* object.

Returns None

Note: For technical reasons, the measure needs to be given via *SetMeasure* method. Otherwise, the Lebesque measure on $[-1, 1]^n$ is assumed.

SetMeasure(*meas*)
Sets the default measure for approximation.

Parameters **meas** – a measure.Measure object

Returns None

fit()
Fits the best curve based on the optional provided orthogonal basis. If no basis is provided, it fits a polynomial of a given degree (at initiation) :return: The fit.

11.4 Sensitivity Analysis

Sensitivity analysis of a dataset based on a fit, sklearn style. The core functionality is provided by SALib .

class sensapprx.**CorrelationThreshold**(*threshold=0.7*)
Selects a minimal set of features based on a given (Pearson) correlation threshold. The transformer omits the maximum number features with a high correlation and makes sure that the remaining features are not correlated behind the given threshold.

Parameters **threshold** – the threshold for selecting correlated pairs.

fit(*X*, *y=None*)
Finds the Pearson correlation among all features, selects the pairs with absolute value of correlation above the given threshold and selects a minimal set of features with low correlation

Parameters

- **X** – Training data
- **y** – Target values (default: *None*)

Returns *self*

fit_transform(*X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

- **X** – numpy array of shape [n_samples, n_features]; Training set.
- **y** – numpy array of shape [n_samples]; Target values.

Returns Transformed array

class sensapprx.**SensAprx**(*n_features_to_select=10*, *regressor=None*, *method='sobol'*, *margin=0.2*, *num_smpl=512*, *num_levels=6*, *grid_jump=1*, *num_resmpl=8*, *reduce=False*, *domain=None*, *probs=None*)
Transform data to select the most secretive factors according to a regressor that fits the data.

Parameters

- **n_features_to_select** – *int* number of top features to be selected
- **regressor** – a sklearn style regressor to fit the data for sensitivity analysis

- **method** – *str* the sensitivity analysis method; defalt 'sobol', other options are 'morris' and 'delta-mmnt'
- **margin** – domain margine, default: .2
- **num_smpl** – number of samples to perform the analysis, default: 512
- **num_levels** – number of levels for morris analysis, default: 6
- **grid_jump** – grid jump for morris analysis, default: 1
- **num_resmpl** – number of resamples for moment independent analysis, default: 10
- **reduce** – whether to reduce the data points to uniques and calculate the averages of the target or not, default: False
- **domain** – pre-calculated unique points, if none, and reduce is *True* then unique points will be found
- **probs** – pre-calculated values associated to *domain* points

fit(*X*, *y*)

Fits the regressor to the data *(X, y)* and performs a sensitivity analysis on the result of the regression.

Parameters

- **X** – Training data
- **y** – Target values

Returns *self*

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

- **X** – numpy array of shape [n_samples, n_features]; Training set.
- **y** – numpy array of shape [n_samples]; Target values.

Returns Transformed array

11.5 Optimized Pipeline Detector

class aml.**AML**(*config=None*, *length=5*, *scoring='accuracy'*, *cat_cols=None*, *surrogates=None*, *min_random_evals=15*, *cv=None*, *check_point='./'*, *stack_res=True*, *stack_probs=True*, *stack_decision=True*, *verbose=1*, *n_jobs=-1*)

A class that accepts a nested dictionary with machine learning libraries as its keys and a dictionary of their parameters and their ranges as value of each key and finds an optimum combination based on training data.

Parameters

- **config** – A dictionary whose keys are scikit-learn-style objects (as strings) and its corresponding values are dictionaries of the parameters and their acceptable ranges/values
- **length** – default=5; Maximum number of objects in generated pipelines
- **scoring** – default='accuracy'; The scoring method to be optimized. Must follow the sklearn scoring signature
- **cat_cols** – default=None; The list of indices of categorical columns

- **surrogates** – default=None; A list of 4-tuples determining surrogates. The first entity of each tuple is a scikit-learn regressor and the 2nd entity is the number of iterations that this surrogate needs to be estimated and optimized. The 3rd is the sampling strategy and the 4th is the *scipy.optimize* solver
- **min_random_evals** – default=15; Number of randomly sampled initial values for hyper parameters
- **cv** – default='ShuffleSplit(n_splits=3, test_size=.25); The cross validation method
- **check_point** – default='./'; The path where the optimization results will be stored
- **stack_res** – default=True; *StackingEstimator's 'res*
- **stack_probs** – default=True; *StackingEstimator's 'probs*
- **stack_decision** – default=True; *StackingEstimator's 'decision*
- **verbose** – default=1; Level of output details
- **n_jobs** – int, default=-1; number of processes to run in parallel

add_surrogate(*estimator*, *itrs*, *sampling=None*, *optim='L-BFGS-B'*)

Adding a regressor for surrogate optimization procedure.

Parameters

- **estimator** – A scikit-learn style regressor
- **itrs** – Number of iterations the *estimator* needs to be fitted and optimized
- **sampling** – default= BoxSample; The sampling strategy (*CompactSample*, *BoxSample* or *SphereSample*)
- **optim** – default='L-BFGS-B';'scipy.optimize' solver

Returns None

eoa_fit(*X*, *y*, ***kwargs*)

Applies evolutionary optimization methods to find an optimum pipeline

Parameters

- **X** – Training data
- **y** – Corresponding observations
- **kwargs** – *EOA* parameters

Returns *self*

fit(*X*, *y*)

Generates and optimizes all legitimate pipelines. The best pipeline can be retrieved from *self.best_estimator_*

Parameters

- **X** – Training data
- **y** – Corresponding observations

Returns *self*

get_top(*num=5*)

Finds the top *n* pipelines

Parameters **num** – Number of pipelines to be returned

Returns An OrderedDict of top models

optimize_pipeline(*seq*, *X*, *y*)
Constructs and optimizes a pipeline according to the steps passed through *seq* which is a tuple of estimators and transformers.

Parameters

- **seq** – the tuple of steps of the pipeline to be optimized
- **X** – numpy array of training features
- **y** – numpy array of training values

Returns the optimized pipeline and its score

types()
Recognizes the type of each estimator to determine proper placement of each

Returns None

class aml.**StackingEstimator**(*estimator*, *res=True*, *probs=True*, *decision=True*)
Meta-transformer for adding predictions and/or class probabilities as synthetic feature(s).

Parameters

- **estimator** – object with fit, predict, and predict_proba methods. The estimator to generate synthetic features from.
- **res** – True (default), stacks the final result of estimator
- **probs** – True (default), stacks probabilities calculated by estimator
- **decision** – True (default), stacks the result of decision function of the estimator

fit(*X*, *y=None*, ***fit_params*)
Fit the StackingEstimator meta-transformer.

Parameters

- **X** – array-like of shape (n_samples, n_features). The training input samples.
- **y** – array-like, shape (n_samples,). The target values (integers that correspond to classes in classification, real numbers in regression).
- **fit_params** – Other estimator-specific parameters.

Returns self, object. Returns a copy of the estimator

set_params(***params*)
Sets the sklearn related parameters for the estimator

Parameters **params** – parameters to be bassed to the estimator

Returns *self*

transform(*X*)
Transform data by adding two synthetic feature(s).

Parameters **X** – numpy ndarray, {n_samples, n_components}. New data, where n_samples is the number of samples and n_components is the number of components.

Returns X_transformed: array-like, shape (n_samples, n_features + 1) or (n_samples, n_features + 1 + n_classes) for classifier with predict_proba attribute; The transformed feature set.

class aml.**Words**(*letters*, *last=None*, *first=None*, *repeat=False*)

This class takes a set as alphabet and generates words of a given length accordingly. A *Words* instant accepts the following parameters:

Parameters

- **letters** – is a set of letters (symbols) to make up the words
- **last** – a subset of *letters* that are allowed to appear at the end of a word
- **first** – a set of words that can only appear at the beginning of a word
- **repeat** – whether consecutive occurrence of a letter is allowed

Generate(*l*)

Generates the set of legitimate words of length *l*

Parameters **l** – int, the length of words

Returns set of all legitimate words of length *l*

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

S

T

U

W