

Documentation for Optimization.py

Yili Yang

June 2017

1 Introduction

The purpose of the optimization is to find the mitigation value in every node of the decision tree that maximizes the current utility

$$x^* = \underset{x}{\operatorname{argmax}} U(x) \quad (1)$$

Our approach to solving this problem is to use the genetic algorithm (GA) combine with a gradient search (GS) method. The GA is used to search the state space globally and to find good initial points for the GS, which applies a gradient descent algorithm to multiple initial points.

2 Genetic Algorithm

2.1 Theory

The GA is an evolutionary algorithm, inspired by the evolution of species in nature. The evolution process starts from a population of vectors with random elements uniformly distributed on $[0, \text{bound}]$. For each generation, the evolution steps are:

1. Select the individuals to perform cross-over and mutation.
2. Cross over among the selected candidate.
3. Mutate result as offspring.
4. Combine the result of offspring and parent together. And selected the top 80 percent of original population amount.
5. Randomly generate 20 percent of original population amount new individuals and combine the above new population.

The mutation and cross-over methods are chosen to fit the optimization problem of the EZ-Climate model.

2.2 GA Class

It is a class that run the genetic search given right population and fit function.

2.2.1 Inputs and Outputs

Inputs:

- **pop_amount** : (int) number of individuals in the population
- **num_feature** : (int) number of elements in each individual, i.e. number of nodes in TreeModel.
- **num_generations** : (int) number of generations of the populations to be evaluated
- **bound** : (float) upper bound of mitigation in each node. (the bound of the initial population. In the model, it would be 1 since the mitigation level won't be larger than 1.)
- **cx_prob** : (float) probability of mating
- **mut_prob** : (float) probability of mutation.
- **utility** : ('Utility' object) object of utility class
- **fixed_values**: (ndarray, optional) nodes to keep fixed
- **fixed_indicies** : (ndarray, optional) indicies of nodes to keep fixed
- **print_progress** : (bool, optional) if the progress of the evolution should be printed

In whcih, the prob of mating and mutation could be changed to provide a faster speed.

Outputs:

the main function that will be use in this class is **run** and it returns the optimization result in a tuple :(final population, the fitness for the final population) the results (individuals) with the highest k fitness will be used in the next step (GS)

2.2.2 Attributes

- **pop_amount** : (int) number of individuals in the population
- **num_feature** : (int) number of elements in each individual, i.e. number of nodes in tree-model
- **num_generations** : (int) number of generations of the populations to be evaluated
- **bound** : (float) upper bound of mitigation in each node
- **cx_prob** : (float) probability of mating
- **mut_prob** : (float) probability of mutation.

- **u** : ('Utility' object) object of utility class
- **fixed_values** : (ndarray, optional) nodes to keep fixed
- **fixed_indicies** : (ndarray, optional) indices of nodes to keep fixed
- **print_progress**: (bool, optional) if the progress of the evolution should be printed

2.2.3 Methods

_generate_population: A private function that generates the initial population **Inputs**:

- **size**: the size of init population

Outputs:

An array of random values with 'size' number of lists. And the lists' length are the number of final nodes. For example, if *size* = 2 and there are four final nodes, the output will be: `array([[0.1,0.4,0.4,0.2],[0.2,0.3,0.3,0.3]])`

```
def _generate_population(self, size):
    """Return 1D-array of random values in the given bound as the initial population"""
    pop = np.random.random([size, self.num_feature])*self.bound
    if self.fixed_values is not None:
        for ind in pop:
            ind[self.fixed_indicies] = self.fixed_values # override
    return pop
```

_evaluate: a private function that evaluate the fitness of the individual within a population. In our model, the fitness is defined by utility.

```
def _evaluate(self, individual):
    """Returns the utility of given individual."""
    return self.u.utility(individual)
```

_select: a private function that returns an array of selected individuals.

Inputs:

- **pop**: the population that you want to make selection from.
- **rate**: the probability of a individual be selected

Outputs: an array of selected individual.

For example:

```
>>> _select([1,2,3,4],0.5)
[1,2,3]
```

```
def _select(self, pop, rate):
    """Returns a 1D-array of selected individuals.
```

Parameters

pop : ndarray

population given by 2D-array with shape ('pop_amount', 'num_features')

rate : float

the probability of an individual being selected

Returns

ndarray

selected individuals

"""

index = np.random.choice(self.pop_amount, int(rate*self.pop_amount), replace=True)

return pop[index,:] #return a list of random instance of pop

_random_index: a private function that generates a random index of individuals of given size (amount · rate). It's is the same with **_select** but just return the index rather than the item itself.

```
def _random_index(self, individuals, size):
```

"""Generate a random index of individuals of size 'size'.

Parameters

individuals : ndarray or list

2D-array of individuals

size : int

number of indices to generate

Returns

ndarray

1D-array of indices

"""

inds_size = len(individuals)

return np.random.choice(inds_size, size)

_selection_tournament: a private function that for given k times, randomly choose a tournament size number of index and pick up the one with the highest fitness. Work flow is as follows:

1. pick a tournament size t
2. randomly choose t individuals in the input population
3. find the one with highest fitness then store the winner

4. repeat until k times

```
def _selection_tournament(self, pop, k, tournsiz, fitness):
    """Select `k` individuals from the input `individuals` using `k`
    tournaments of `tournsiz` individuals.

    Parameters
    -----
    individuals : ndarray or list
        2D-array of individuals to select from
    k : int
        number of individuals to select
    tournsiz : int
        number of individuals participating in each tournament
    fitness :
        utility in out model

    Returns
    -----
    ndarray s
        selected individuals

    """
    chosen = []
    # for k times, randomly choose a tournsiz number of index and pick up the
    for i in xrange(k):
        index = self._random_index(pop, tournsiz)
        aspirants = pop[index]
        aspirants_fitness = fitness[index]
        chosen_index = np.where(aspirants_fitness == np.max(aspirants_fitness))[0]
        if len(chosen_index) != 0:
            chosen_index = chosen_index[0]
        chosen.append(aspirants[chosen_index])
    return np.array(chosen)
```

_two_point_cross_over: a private function that does the two-point cross-over. Two-point crossover calls for two points to be selected on the parent organism strings. Everything between the two points is swapped between the parent organisms, rendering two child organisms. A two-point cross-over is illustrated as following graph:

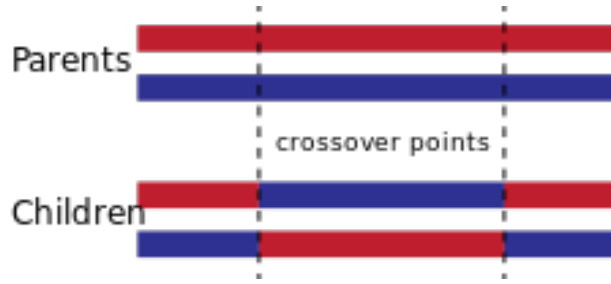


Figure 1: Two Point Crossover

```
def _two_point_cross_over(self, pop):
    """Performs a two-point cross-over of the population.

    Parameters
    -----
    pop : ndarray
        population given by 2D-array with shape ('pop_amount', 'num_fe

    """
    child_group1 = pop[:, ::2] # instance with even index
    child_group2 = pop[:, 1::2] # instance with odd index
    for child1, child2 in zip(child_group1, child_group2):
        if np.random.random() <= self.cx_probab:
            #generates 2 random index for the swap, can be done mu
            cxpoint1 = np.random.randint(1, self.num_feature)
            cxpoint2 = np.random.randint(1, self.num_feature - 1)
            if cxpoint2 >= cxpoint1:
                cxpoint2 += 1
            else: # Swap the two cx points
                cxpoint1, cxpoint2 = cxpoint2, cxpoint1
            child1[cxpoint1:cxpoint2], child2[cxpoint1:cxpoint2] \
                = child2[cxpoint1:cxpoint2].copy(), child1[cxpoint1:cxpo
            if self.fixed_values is not None:
                child1[self.fixed_indicies] = self.fixed_values
                child2[self.fixed_indicies] = self.fixed_values
```

_uniform_cross_over: a private function that does the uniform cross-over. The uniform crossover uses a fixed mixing ratio between two parents. Unlike single- and two-point crossover, the uniform crossover enables the parent chromosomes to contribute the gene level rather than the segment level.

If the mixing ratio is 0.5, the offspring has approximately half of the genes from first parent and the other half from second parent, although cross over points can be randomly chosen as seen below:



Figure 2: Uniform Crossover

```
def _uniform_cross_over(self, pop, ind_prob):
    """Performs a uniform cross-over of the population.

    Parameters
    -----
    pop : ndarray
        population given by 2D-array with shape ('pop_amount', 'num_features')
    ind_prob : float
        probability of feature cross-over

    """
    child_group1 = pop[:, :2]
    child_group2 = pop[1::2, :2]
    for child1, child2 in zip(child_group1, child_group2):
        size = min(len(child1), len(child2))
        for i in range(size):
            if np.random.random() < ind_prob:
                child1[i], child2[i] = child2[i], child1[i]
```

_mutate: A private function that mutates individual's elements. The mutation is constrained by a input parameter called **scale**. And the formula for mutation is as follows: $pop[i][j] = \max\{0.0, pop[i][j] + (RandomNumber - 0.5) \cdot scale\}$ where $pop[i][j]$ is the element that is been mutated

```
def _mutate(self, pop, ind_prob, scale=2.0):
    """Mutates individual's elements. The individual has a probability of
    beeing selected and every element in this individual has a probability
    mutated. The mutated value is a random number.

    Parameters
    -----
    pop : ndarray
        population given by 2D-array with shape ('pop_amount', 'num_features')
    ind_prob : float
        probability of feature mutation
    scale : float
        scaling constant of the random generated number for mutation

    """
```

```

# it is using a expectation of prob. Can be done much better.
pop_tmp = np.copy(pop)
mutate_index = np.random.choice(self.pop_amount, int(self.mut_prob * self.pop_amount))
for i in mutate_index:
    feature_index = np.random.choice(self.num_feature, int(ind_prob * self.num_feature))
    for j in feature_index:
        if self.fixed_indicies is not None and j in self.fixed_indicies:
            continue
        else:
            pop[i][j] = max(0.0, pop[i][j] + (np.random.random() - 0.5) * scale)

```

_uniform_mutation: A private function that performance the uniform mutation which means the mutated value is the current value plus a scaled uniform $[-0.5, 0.5]$ random value.

```

def _uniform_mutation(self, pop, ind_prob, scale=2.0):
    """Mutates individual's elements. The individual has a probability of
    beeing selected and every element in this individual has a probability
    mutated. The mutated value is the current value plus a scaled uniform
    random value.

    Parameters
    -----
    pop : ndarray
        population given by 2D-array with shape ('pop_amount', 'num_features')
    ind_prob : float
        probability of feature mutation
    scale : float
        scaling constant of the random generated number for mutation

    """
    pop_len = len(pop)
    mutate_index = np.random.choice(pop_len, int(self.mut_prob * pop_len), replace=True)
    for i in mutate_index:
        prob = np.random.random(self.num_feature)
        inc = (np.random.random(self.num_feature) - 0.5) * scale
        pop[i] += (prob > (1.0 - ind_prob)).astype(int) * inc
        pop[i] = np.maximum(1e-5, pop[i])
        if self.fixed_values is not None:
            pop[i][self.fixed_indicies] = self.fixed_values

```

_show_evolution: print statistics of the evolution of the population

```

def _show_evolution(self, fits, pop):
    """Print statistics of the evolution of the population."""
    length = len(pop)
    mean = fits.mean()
    std = fits.std()
    min_val = fits.min()

```



```

max_val = fits.max()
print (" Min {} \n Max {} \n Avg {}".format(min_val, max_val, mean))
print (" Std {} \n Population Size {}".format(std, length))
print (" Best Individual: ", pop[np.argmax(fits)])

```

_survive: the 80 percent of the individuals with best fitness survives to the next generation.

```

def _survive(self, pop_tmp, fitness_tmp):
    """The 80 percent of the individuals with best fitness survives to
    the next generation.

    Parameters
    -----
    pop_tmp : ndarray
               population
    fitness_tmp : ndarray
               fitness values of `pop_tmp`

    Returns
    -----
    ndarray
        individuals that survived

    """
    index_fits = np.argsort(fitness_tmp)[::-1]
    fitness = fitness_tmp[index_fits]
    pop = pop_tmp[index_fits]
    num_survive = int(0.8*self.pop_amount)
    survive_pop = np.copy(pop[:num_survive])
    survive_fitness = np.copy(fitness[:num_survive])
    return np.copy(survive_pop), np.copy(survive_fitness)

```

run: main function of the GA class. Return the (final population, the fitness for the final population). The whole algo follows these steps:

1. Select the individuals to perform cross-over and mutation.
2. Cross over among the selected candidate.
3. Mutate result as offspring.
4. Combine the result of offspring and parent together. And selected the top 80 percent of original population amount.
5. Random Generate 20 percent of original population amount new individuals and combine the above new population.

Within the process, the base model is doing uniform cross over with rate 0.5 and uniform mutation with rate 0.25 but the parameters can be changed and it can be change to random

cross-over or mutation.

```
def run(self):
    """
    Returns
    -----
    tuple
        final population and the fitness for the final population

    Note
    ----
    Uses the :mod:`~multiprocessing` package.

    """
    print("-----Genetic Evolution Starting-----")
    pop = self._generate_population(self.pop_amount)
    pool = multiprocessing.Pool(processes=multiprocessing.cpu_count())
    fitness = pool.map(self._evaluate, pop) # how do we know pop[i] belongs
    fitness = np.array([val[0] for val in fitness])
    u_hist = np.zeros(self.num_gen) # not been used ...
    for g in range(0, self.num_gen):
        print("-- Generation {} --".format(g+1))
        pop_select = self._select(np.copy(pop), rate=1)

        self._uniform_cross_over(pop_select, 0.50)
        self._uniform_mutation(pop_select, 0.25, np.exp(-float(g)/self.n
        #self._mutate(pop_select, 0.05)

        fitness_select = pool.map(self._evaluate, pop_select)
        fitness_select = np.array([val[0] for val in fitness_select])

        pop_tmp = np.append(pop, pop_select, axis=0)
        fitness_tmp = np.append(fitness, fitness_select, axis=0)

        pop_survive, fitness_survive = self._survive(pop_tmp, fitness_tm

        pop_new = self._generate_population(self.pop_amount - len(pop_sur
        fitness_new = pool.map(self._evaluate, pop_new)
        fitness_new = np.array([val[0] for val in fitness_new])

        pop = np.append(pop_survive, pop_new, axis=0)
        fitness = np.append(fitness_survive, fitness_new, axis=0)
        if self.print_progress:
            self._show_evolution(fitness, pop)
        u_hist[g] = fitness[0]
```

```

fitness = pool.map(self._evaluate, pop)
fitness = np.array([val[0] for val in fitness])
return pop, fitness

```

3 Gradient Search

3.1 Theory

The GS uses the gradient descent algorithm and the numerical gradient to find the optimal mitigation points. Moreover, it uses the Adaptive Moment Estimation (Adam) learning rate together with an accelerator scaler to update the points. Adam is a method that computes adaptive learning rates for each parameter.

In addition to storing an exponentially decaying average of past squared gradients, Adam also keeps an exponentially decaying average of past gradients. The accelerator is used to amplify low gradient values of mitigation values in nodes in the end of the tree, and thus reduce computation time. The **run** method takes `initial_point_list` and `topk` as arguments, runs the gradient descent optimization of the `topk` first elements of the `initial_point_list`, and picks the resulting point with the highest utility.

3.2 GS Class

Gradient search optimization algorithm for the EZ-Climate model

3.2.1 Inputs and Outputs

Inputs:

- **utility** : ('Utility' object) object of utility class
- **learning_rate** : (float) starting learning rate of gradient descent
- **var_nums** : (int) number of elements in array to optimize
- **accuracy** : (float) stop value for the gradient descent
- **fixed_values** : (ndarray, optional) nodes to keep fixed
- **fixed_indicies** : (ndarray, optional) indices of nodes to keep fixed
- **print_progress** : (bool, optional) if the progress of the evolution should be printed
- **scale_alpha** : (ndarray, optional) array to scale the learning rate

Output:

The main output is from the **run** function same as the GA class. The function returns a tuple: (best mitigation point, the utility of the best mitigation point)

3.2.2 Attributes

- **utility** : ('Utility' object) object of utility class
- **learning_rate** : (float) starting learning rate of gradient descent
- **var_nums** : (int) number of elements in array to optimize
- **accuracy** : (float) stop value for the gradient descent
- **fixed_values** : (ndarray, optional) nodes to keep fixed
- **fixed_indicies** : (ndarray, optional) indicies of nodes to keep fixed
- **print_progress** : (bool, optional) if the progress of the evolution should be printed
- **scale_alpha** : (ndarray, optional) array to scale the learning rate

The attributes are the same as the inputs. All of this paramters is used to do the gradient search.

3.2.3 Methods

_partial_grad: A private function to calculate the gradient on a certain node. Gradient here is:

$$gradient = \frac{f(x + \Delta) - f(x - \Delta)}{2\Delta} \quad (2)$$

where f(x) is the general fitness function for the GA Class. For the base case, it is the utility function. **Input**:

- **i**: the index of the node in the utility array.

Output:

- **grad**: the the gradient of the i the node
- **i**: the index of the node in the utility array.

```
def _partial_grad(self, i):  
    """Calculate the ith element of the gradient vector."""  
    m_copy = self.m.copy()  
    m_copy[i] = m_copy[i] - self.delta if (m_copy[i] - self.delta)>=0 else 0  
    minus_utility = self.u.utility(m_copy)  
    m_copy[i] += 2*self.delta  
    plus_utility = self.u.utility(m_copy)
```

```

grad = (plus_utility-minus_utility) / (2*self.delta) # the math is trivial
return grad, i

```

numerical_gradient: calculate utility gradient for each node.

```

def numerical_gradient(self, m, delta=1e-08, fixed_indicies=None):
    """Calculate utility gradient numerically.

    Parameters
    -----
    m : ndarray or list
        array of mitigation
    delta : float, optional
        change in mitigation
    fixed_indicies : ndarray or list, optional
        indicies of gradient that should not be calculated

    Returns
    -----
    ndarray
        gradient

    """
    self.delta = delta
    self.m = m
    if fixed_indicies is None:
        fixed_indicies = []
    grad = np.zeros(len(m))
    if not isinstance(m, np.ndarray):
        self.m = np.array(m)
    pool = multiprocessing.Pool()
    indicies = np.delete(range(len(m)), fixed_indicies)
    res = pool.map(self._partial_grad, indicies)
    for g, i in res:
        grad[i] = g
    pool.close()
    pool.join()
    del self.m
    del self.delta
    return grad

```

_accelerate_scale: An empirical accelerate function which does the following modification:

1. if the search direction remains the same, accelerate the search by 1.1 times speed.
2. otherwise, reset the search speed.

The accelerate ratio 1.1 is purely empirical and can be modified in the latter research.

```

def _accelerate_scale(self, accelerator, prev_grad, grad):
    sign_vector = np.sign(prev_grad * grad)
    scale_vector = np.ones(self.var_nums) * ( 1 + 0.10)
    accelerator[sign_vector <= 0] = 1
    accelerator *= scale_vector
    return accelerator

```

gradient_descent: main body of the gradient search algorithm (Adam). In addition to storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t , similar to momentum:

$$\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2
\end{aligned}$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1, can be modified for latter research).

We counteract these biases by computing bias-corrected first and second moment estimates:

$$\begin{aligned}
\hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1 - \beta_2^t}
\end{aligned}$$

We then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \quad (3)$$

In this case, we also add a acceleration coefficient which is define by above function to accelerate the searching process:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \times \text{accelerator} \quad (4)$$

Finally, we get the item in the search results with the highest utility.

```

def gradient_descent(self, initial_point, return_last=False):
    """Gradient descent algorithm. The `initial_point` is updated using the
    Adam algorithm. Adam uses the history of the gradient to compute indiv
    step sizes for each element in the mitigation vector. The vector of st
    sizes are calculated using estimates of the first and second moments o
    the gradient.

    Parameters

```

```

-----
initial_point : ndarray
    initial guess of the mitigation
return_last : bool, optional
    if True the function returns the last point, else the point
    with highest utility

Returns
-----
tuple
    (best point, best utility)

"""
num_decision_nodes = initial_point.shape[0]
x_hist = np.zeros((self.iterations+1, num_decision_nodes))
u_hist = np.zeros(self.iterations+1)
u_hist[0] = self.u.utility(initial_point)
x_hist[0] = initial_point

beta1, beta2 = 0.90, 0.90
eta = 0.0015 # learning rate
eps = 1e-3
m_t, v_t = 0, 0

prev_grad = 0.0
accelerator = np.ones(self.var_nums)
# formula at http://sebastianruder.com/optimizing-gradient-descent/index.html
for i in range(self.iterations):
    grad = self.numerical_gradient(x_hist[i], fixed_indicies=self.fixed_indicies)
    m_t = beta1*m_t + (1-beta1)*grad
    v_t = beta2*v_t + (1-beta2)*np.power(grad, 2)
    m_hat = m_t / (1-beta1**(i+1))
    v_hat = v_t / (1-beta2**(i+1))
    if i != 0:
        accelerator = self._accelerate_scale(accelerator, prev_grad)

    new_x = x_hist[i] + ((eta*m_hat)/(np.square(v_hat)+eps)) * accelerator
    new_x[new_x < 0] = 0.0

    if self.fixed_values is not None:
        new_x[self.fixed_indicies] = self.fixed_values

    x_hist[i+1] = new_x
    u_hist[i+1] = self.u.utility(new_x)[0]
    prev_grad = grad.copy()

```

```

        if self.print_progress:
            print("-- Iteration {} -- \n Current Utility: {}".format(
                iteration, new_utility))
            print(new_x)

    if return_last:
        return x_hist[i+1], u_hist[i+1]
    best_index = np.argmax(u_hist)
    return x_hist[best_index], u_hist[best_index]

```

run: run the GS on the init point (provided by GA) and get the optimized result.

```

def run(self, initial_point_list, topk=4):
    """Initiate the gradient search algorithm.

    Parameters
    -----
    initial_point_list : list
        list of initial points to select from
    topk : int, optional
        select and run gradient descent on the `topk` first points of
        `initial_point_list`

    Returns
    -----
    tuple
        best mitigation point and the utility of the best mitigation point

    Raises
    -----
    ValueError
        If `topk` is larger than the length of `initial_point_list`.

    Note
    ----
    Uses the :mod:`~multiprocessing` package.

    """
    print("-----Gradient Search Starting-----")

    if topk > len(initial_point_list):
        raise ValueError("topk {} > number of initial points {}".format(
            topk, len(initial_point_list)))

    candidate_points = initial_point_list[:topk]
    mitigations = []
    utilities = np.zeros(topk)

```



```

for cp, count in zip(candidate_points, range(topk)):
    if not isinstance(cp, np.ndarray):
        cp = np.array(cp)
    print("Starting process {} of Gradient Descent".format(count+1))
    m, u = self.gradient_descent(cp)
    mitigations.append(m)
    utilities[count] = u
best_index = np.argmax(utilities)
return mitigations[best_index], utilities[best_index]

```

4 Coordinate Descent

Coordinate descent optimization algorithm for the EZ-Climate model.

4.1 Theory

Coordinate descent is based on the idea that the minimization of a multivariate function $F(x)$ can be achieved by minimizing it along one direction at a time, i.e., solving univariate (or at least much simpler) optimization problems in a loop. In the simplest case of cyclic coordinate descent, one cyclically iterates through the directions, one at a time, minimizing the objective function with respect to each coordinate direction at a time.

4.2 Coordinate Descent

The function is running **fmin** from scipy package for every single node at a time and cyclically iterates through the nodes.

```

class CoordinateDescent(object):
    """Coordinate descent optimization algorithm for the EZ-Climate model.

    Parameters
    -----
    utility : `Utility` object
               object of utility class
    var_nums : int
               number of elements in array to optimize
    accuracy : float
               stop value for the utility increase
    iterations : int
               maximum number of iterations

    Attributes

```

```

-----
utility : `Utility` object
          object of utility class
var_nums : int
          number of elements in array to optimize
accuracy : float
          stop value for the utility increase
iterations : int
          maximum number of iterations

"""
def __init__(self, utility, var_nums, accuracy=1e-4, iterations=100):
    self.u = utility
    self.var_nums = var_nums
    self.accuracy = accuracy
    self.iterations = iterations

def _min_func(self, x, m, i):
    m_copy = m.copy()
    m_copy[i] = x
    return -self.u.utility(m_copy)[0]

def _minimize_node(self, node, m):
    from scipy.optimize import fmin
    return fmin(self._min_func, x0=m[node], args=(m, node), disp=False)

def run(self, m):
    """Run the coordinate descent iterations.

    Parameters
    -----
    m : initial point

    Returns
    -----
    tuple
        best mitigation point and the utility of the best mitigation p

    Note
    ----
    Uses the :mod:`~scipy` package.

    """
    num_decision_nodes = m.shape[0]
    x_hist = []

```

```

u_hist = []
nodes = range(self.var_nums)
x_hist.append(m.copy())
u_hist.append(self.u.utility(m)[0])
print("-----Coordinate Descent Starting-----")
print("Starting Utility: {}".format(u_hist[0]))
for i in range(self.iterations):
    print("-- Iteration {} --".format(i+1))
    node_iteration = np.random.choice(nodes, replace=False, size=len(nodes))
    for node in node_iteration:
        m[node] = max(0.0, self._minimize_node(node, m))
    x_hist.append(m.copy())
    u_hist.append(self.u.utility(m)[0])
    print("Current Utility: {}".format(u_hist[i+1]))
    if np.abs(u_hist[i+1] - u_hist[i]) < self.accuracy:
        break
return x_hist[-1], u_hist[-1]

```