GenAlgNB

November 20, 2018

In [2]:
```
import numpy as np
from string import ascii_letters, punctuation

class GenAlgorithmString(object):

    def __init__(self,symbols = (ascii_letters + punctuation + ' '),n_population=10,
                 n_generation = 100,method = 'roulette',K=4,desired_fitness=0.4):
        """
        This is implementation of genetic algorithm, prepared for MD seminary
        presentation, it tries to generate given input by simulating evolutio
        n mechanisms.

        I used sklearn convention in naming fundamental part of it, so by cal
        ling fit method one can fits the word/sentence and by calling transfo
        rm one can activates algorithm so it tries to find the best (most sim
        ilar) string.

        Attributes

        symbols
        a set of characters from which algorithm should have created string,
        by default it is a set of asii letters and punctuation signs

        n_population
        the quantity of individuals for each population

        n_generation
        in how many generations algorithm should have found the best individu
        al

        method
        which selection method should be used for selecting individuals, by
        default it is roulette, but also ranking could be used.

        K
        how many best individuals should be used for pairing
```

```
        desired_fitness
        threshold that must be achieved to stop algorithm (ver 2.1)

        """

        self = self
        self.symbols = symbols
        self.n_population = n_population
        self.n_generation = n_generation
        self.method = method
        self.K = K
        self.desired_fitness = desired_fitness

    def get_symbol(self):
        """ Generates random symbol """
        return self.symbols[np.random.randint(len(self.symbols))]

    def _generate_population(self):
        """ Generates array for population """
        self.population = np.chararray((self.n_population,
                                        self.n_genotype),unicode=True)

    def fit(self,aim):
        """ Fits the given sentence to the model """
        target = np.chararray((len(aim)),unicode=True)
        for chunk in range(len(target)):
            target[chunk] = aim[chunk]
        self.target = target
        self.n_genotype = len(target)
        self._generate_population()

    # def transform(self):
    #     """ Performs the whole algorithm """
    #     self._mutate_population()
    #     for generation in range(self.n_generation):
    #         self.descendants_generation()

    def transform(self):
        """ Performs the whole algorithm until desired fitness is reached. """
        self._mutate_population()
        pop = np.max(self._population_fitness())
        self.n_generation = 0
        while (pop < self.desired_fitness):
            self.descendants_generation()
            self.n_generation += 1
            pop = np.max(self._population_fitness())

    def _pooling(self):
```

```python
        """ Mutates the whole chromosome, i.e. generates random set of
        characters"""
        chromosome = np.chararray((self.n_genotype),unicode=True)
        for locus in range(self.n_genotype):
            chromosome[locus] = self.get_symbol()
        return chromosome

    def _mutate_population(self):
        """ Mutates the whole population, for each individual performs
        pooling """
        for individual in range(self.n_population):
            self.population[individual] = self._pooling()

    @staticmethod
    def _check_fitness(chromosome,target):
        """ Checks the fitness of individual """
        return np.count_nonzero(chromosome[chromosome == target])/len(chromosome)

    @staticmethod
    def _pairing(parents):
        """ Method for pairing chromosomes and generating descendants, array of
        characters with shape [2,n_genotype] """
        children = np.chararray((2,parents.shape[1]),unicode=True)
        n_heritage = np.random.randint(0,parents[0].shape[0])
        children[0] = np.concatenate([parents[0][:n_heritage],
                                      parents[1][n_heritage:]])
        children[1] = np.concatenate([parents[1][:n_heritage],
                                      parents[0][n_heritage:]])
        return children

    def _population_fitness(self):
        """ Checks the fitness of each individual in population """
        fitness = np.zeros(self.population.shape[0])
        for individual in range(self.population.shape[0]):
            fitness[individual] = self._check_fitness(self.population[individual],
                                                      self.target)
        return fitness

    def _ranking(self):
        """ Ranking method for individuals selection """
        fitness = self._population_fitness()
        population = self.population.copy()
        population_of_best = np.chararray((self.K,self.population.shape[1]),
                                          unicode=True)
        if np.any(fitness != 0):
            for k in range(self.K):
                population_of_best[k] = population[np.where(np.max(fitness))]
                population = np.delete(population,np.where(np.max(fitness)),0)
```

```
            return population_of_best
        else:
            return mutate_population(population)

    def random_mutation(self):
        """ Randomly mutate population """
        N,C = self.population.shape
        new_population = self.population.copy()
        n_mutations = round(self.population.size * 0.02)
        for n in range(n_mutations):
            new_population[np.random.randint(N),
                           np.random.randint(C)] = self.get_symbol()
        self.population = new_population

    def descendants_generation(self):
        """ Generaters new population from pairing old one """
        fitness = self._population_fitness()
        if np.any(fitness != 0):
            if self.method == 'ranking':
                bests = self._ranking()
            elif self.method == 'roulette':
                bests = self.roulette()
            self._mutate_population()
            for n in range(round(self.K/2)):
                parent1 = bests[np.random.randint(self.K)]
                parent2 = bests[np.random.randint(self.K)]
                descendants = self._pairing(np.array([parent1,parent2]))
                self.population[(n*2)] = descendants[0].squeeze()
                self.population[(n*2)+1] = descendants[1].squeeze()
                # self.random_mutation()
        else:
            self._mutate_population()


    def roulette_wheel(self):
        """ Generates roulette wheel based on fitness of each individual.
            Needed for roulette selection of individuals"""
        generation = self._population_fitness()
        probability = 0
        wheel = np.zeros(3)
        if np.any(generation != 0):
            for individual in range(self.n_population):
                if generation[individual] > 0:
                    ind_probability = probability + (
                    generation[individual] / np.sum(generation))
                    wheel = np.vstack([wheel,[individual,
                                       probability,ind_probability]])
                    probability = probability + (
```

```
                                    generation[individual] / np.sum(generation))
                return wheel[1:,:]

        @staticmethod
        def roulette_swing(wheel):
            """ Swings the roulette wheel.
                Needed for roulette selection of individuals """
            which = np.random.random()
            for n in range(len(wheel)):
                if which > wheel[n][1] and which < wheel[n][2]:
                    return int(wheel[n][0])

        def roulette(self):
            """ Performs roulette selection on population """
            wheel = self.roulette_wheel()
            winners = np.chararray((self.K,self.population.shape[1]),unicode=True)
            for n in range(self.K):
                which = self.roulette_swing(wheel)
                winners[n] = self.population[which]
            return winners

    if __name__ == '__main__':
        genalg = GenAlgorithmString(symbols = (ascii_letters),n_generation = 0,
                                    K=5,n_population=10,method = 'roulette',
                                    desired_fitness = 0.4)
        genalg.fit('Programming is awesome')
        genalg.transform()
        max_fitness = np.max(genalg._population_fitness())
        best_individual = genalg.population[np.where(max_fitness)]
        print('Number of generations: %i \nBest fitness: %f' %(genalg.n_generation,
        max_fitness))
```

```
Number of generations: 293427
Best fitness: 0.409091
```

In [ ]: