Assignment_4_Populations_[netid]

September 21, 2021

1 Assignment 4: Populations

In our last assignment, we used systematic experimentation to figure out how and why our hill-climber was working – paying particular attention to how we modify selection pressure, and how that affects search trajectories over fitness landscapes of varying ruggedness.

In this assignment, we'll continue to build out the basic backbone of our evolutionary algorithm, while applying the same lens of systematic investigation and exploration around implementation details. In particular, we'll expand our single-parent/single-child hillclimber into a full population, and explore how crossover and selection manifest themselves in our algorithm.

[ ]:
```
# imports
import numpy as np
import copy
import matplotlib.pyplot as plt
plt.style.use('seaborn')

import scikits.bootstrap as bootstrap
import warnings
warnings.filterwarnings('ignore') # Danger, Will Robinson! (not a scalable␣
 ↪hack, and may surpress other helpful warning other than for ill-conditioned␣
 ↪bootstrapped CI distributions)

import scipy.stats # for finding statistical significance
```

1.0.1 Q1: Implementing Individuals within a Population

As we beging to work with populations, it will get a increasingly messy to keep track of each individual's genome and fitness seperately as they move around the population and through generational time. To help simplify this, let's implement each individual within a population as an instance of an `Individual` class. To start, this class will be quite simple and will just be an object which has attributes for both the individual's `genome` and its `fitness`. Since we will only be using fitness functions that depend on a single individual in this assignment, let's also implement an `eval_fitness` for each individual that will evaluate and update its stored fitness value when called.

[ ]:
```
class Individual:

    def __init__(self, fitness_function, bit_string_length):
        ...

    def eval_fitness(self):
        ...
```

1.0.2 Q2: Modifying the hillclimber

Let's take the basic hillclimber from our last assignment and turn it into a full fleged evolutionary algorithm. Again, please feel free to leverage your prior work (or our prior solution sets) and copy-and-paste liberally.

In particular, our first version of this algorithm will have a number of parents and a number of children given as parameters (a la evolutionary strategies), two-point crossover (of randomly selected parents), and truncation selection. Please also include arguemtns to this evolutionary_algorithm function which allow you dictate whether the algorithm will use mutation (the same single bit flip we used before), crossover, or both (for use in the following question).

To get a finer-grain look at convergence rates of these different approaches, let's also modify the output of this function to return the fitness of the top individual at each generation.

[ ]:
```
def evolutionary_algorithm(fitness_function=None, total_generations=100,␣
↪num_parents=10, num_children=10, bit_string_length=10,␣
↪num_elements_to_mutate=1, crossover=True):
    """ Evolutinary Algorithm (copied from the basic hillclimber in our last␣
↪assignment)

        parameters:
        fitness_funciton: (callable function) that return the fitness of a␣
↪genome
                          given the genome as an input parameter (e.g. as␣
↪defined in Landscape)
        total_generations: (int) number of total iterations for stopping␣
↪condition
        num_parents: (int) the number of parents we downselect to at each␣
↪generation (mu)
        num_childre: (int) the number of children (note: parents not included␣
↪in this count) that we baloon to each generation (lambda)
        bit_string_length: (int) length of bit string genome to be evoloved
        num_elements_to_mutate: (int) number of alleles to modify during␣
↪mutation (0 = no mutation)
        crossover (bool): whether to perform crossover when generating children

        returns:
```

```
        fitness_over_time: (numpy array) track record of the top fitness value␣
↪at each generation
    """

    # initialize record keeping
    ...

    # the initialization proceedure
    ...

    # get population fitness
    ...

    ... # repeat

        # the modification procedure
        ...

            # inheretance
            ...

            # crossover
            ...

            # mutation
            ...

        # the assessement procedure
        ...

        # selection procedure
        ...

        # record keeping
        ...

    return ...
```

1.0.3 Q3: Running Experiments

Similar to last week, let's systemtically run and plot the results. To start let's use `50` parents (mu) and `50` children (lambda). For simplicity, let's go back to the one-max problem (and normalize the fitness, using `np.mean` isntead of `np.sum` for our fitness function in case we want to make comparisons across different genome lengths – though for now, let's start with a bit string genome of length `200`).

Also taking pieces from your experimental comparison scripts from last week, please run this for

the the case of mutation only, crossover only, and employing both mutation and crossover. Run 20 independent repitions for each condition.

[ ]:
```
num_runs = 20
total_generations = 100
num_elements_to_mutate = 1
bit_string_length = 200
num_parents = 50
num_children = 50
experiment_results = {}

...
```

1.0.4 Q3b: Visualization

We will also modify our plotting scripts from before to show how fitness increases over generational time across these three treatments (with boostrapped confidence intervals as before). As we also did previously, please plot the three experimental conditions run above on the same figure for ease of comparisons.

[ ]:
```
def plot_mean_and_bootstrapped_ci_over_time(input_data = None, name = "change␣
↪me", x_label = "change me", y_label="change me", y_limit = None):
    """

    parameters:
    input_data: (numpy array of shape (max_k, num_repitions)) solution metric␣
↪to plot
    name: (string) name for legend
    x_label: (string) x axis label
    y_label: (string) y axis label

    returns:
    None
    """

    ...
```

1.0.5 Q4: Analysis of Crossover

Is crossover effective on this problem? How does crossover compare to mutation? How do the two interact?

insert text here

1.0.6 Q5: Propose and Implement a New Crossover Modification

We've implemented one specfic type of crossover (two-point crossover with two randomly chosen parents). What other variatons to crossover might you consider? Describe it in the box below, and what you anticipate as the effects of it (positive or negative).

insert text here

1.0.7 Q5b: Let's test it!

Copy your evoluationary_algoirthm code and modify it to include your new experimental treatment. Run and visualize this treatment as above. Feel free to alo pull in any statistical test scripts/functions from last week, should that help you to analyze and compare this new approach.

[ ]:
```
# your new evolutionary_algorithm
...
```

[ ]:
```
# experimentation
...
```

[ ]:
```
# visualization
...
```

1.0.8 Q6: Well... What happened?

Describe the effect of your approach. If it did not work out as expected, please hypotheize as to why this is the case. If it did work out well, please comment on how broadly you think this finding might apply (or in what experimental conditions you might expect to come to a different conclusion).

insert text here

1.0.9 Q7: Implementing Tournament Selection

Aside from crossover, including populations also gives us the opportunity to explore alternate selection mechanisms. As mentioned in class, tournament selection is one of my go-to methods for parent selection, so let's implement it here. The tournament should rely on input parameters such as the `tournament_size` to determine how many solutions will compete in a given tournament or `num_tournament_winners` to determine how many individuals from each tournament will be selected to move on as parents of the next generation. Touraments can be selected from the population with or without replacement (specifically I'm referring to making sure all individuals appear in at least one tournament before any individual partakes in one for a second time), and here feel free to use whichever version is simpler for you to implement and understand (which I expect will be the case with replacement).

```
[ ]: def evolutionary_algorithm(fitness_function=None, total_generations=100,␣
     ↪num_parents=10, num_children=10, bit_string_length=10,␣
     ↪num_elements_to_mutate=1, crossover=True, tournament_size=4,␣
     ↪num_tournament_winners=2):
         """ Evolutinary Algorithm (copied from the basic hillclimber in our last␣
     ↪assignment)

             parameters:
             fitness_funciton: (callable function) that return the fitness of a␣
     ↪genome
                               given the genome as an input parameter (e.g. as␣
     ↪defined in Landscape)
             total_generations: (int) number of total iterations for stopping␣
     ↪condition
             num_parents: (int) the number of parents we downselect to at each␣
     ↪generation (mu)
             num_childre: (int) the number of children (note: parents not included␣
     ↪in this count) that we baloon to each generation (lambda)
             bit_string_length: (int) length of bit string genome to be evoloved
             num_elements_to_mutate: (int) number of alleles to modify during␣
     ↪mutation (0 = no mutation)
             crossover: (bool) whether to perform crossover when generating children
             tournament_size: (int) number of individuals competing in each␣
     ↪tournament
             num_tournament_winners: (int) number of individuals selected as future␣
     ↪parents from each tournament (must be less than tournament_size)

             returns:
             fitness_over_time: (numpy array) track record of the top fitness value␣
     ↪at each generation
         """

         ...
```

1.0.10 Q8: Run and Plot

We discussed in class that the number of individuals participating in a tournament affects the amount of selection pressure it produces, presumably the same is true for the number of individuals selected from that tournament. So let's play around and generate some data to try and get to the bottom of it! In particular, let's run the following four experimental conditions: `10 select 5`, `20 select 10`, `20 select 5`, `50 select 10` (where the first number is how many individuals are in a tournament, and the second number is how many are selected from that tournament). Let's run these on the full-fledged evolutionary_algorithm including both mutation and crossover (for consistency and ease of grading please the original evolutionary algorithm implementation from `Q2` rather than your new implementation in `Q5` by either rerunning the prior code block, or by coping

and pasting it in a new code block below). As above, please visualize the resulting fitnes over time and their boostrapped confidence intervals as well.

[ ]:
```
# if wanting to copy the original evolutionary_algorithm implementation here (e.
↪g. so you can run Kernel -> Restart & Run All without having to manually␣
↪rerun the block above within that)
...
```

[ ]:
```
num_runs = 20
total_generations = 100
num_elements_to_mutate = 1
bit_string_length = 200
num_parents = 50
num_children = 50
experiment_results = {}

...
```

[ ]:
```
# plotting
...
```

1.0.11 Q9: Analysis

What do these results suggest matter about the values of the tournnament size and the number of winners selected? Is this suprising?

insert text here

1.0.12 Q10: Future Work

Again, we've just scratched the tip of the iceberg in terms of understanding or efficiently employing populations in evolutionary algorithms. If you were to run one more experiment here (i.e. another question in this assignment) what would you test next? If you were to deeply investigate some phenomenon around populations/selection/crossover (i.e. spend 6 weeks on a course project) what broader topic might you dig into?

insert text here

1.0.13 Congratulations, you made it to the end!

Nice work – and hopefully you're starting to get the hang of these!

Please save this file as a .ipynb, and also download it as a .pdf, uploading **both** to blackboard to complete this assignment.

For your submission, please make sure that you have renamed this file (and that the resulting pdf follows suit) to replce `[netid]` with your UVM netid. This will greatly simplify our grading pipeline, and make sure that you receive credit for your work.

Academic Integrity Attribution During this assignment I collaborated with:

insert text here