Assignment 7: Novelty Search

In our last assignment, we explored the idea of measuring diversity. This week we'll turn it up to eleven, and directly incentivize diversity by playing around with novelty search.

While not quite a deceptive landscape, we'll see how novelty search interacts with a rugged fitness landscape by revisiting our prior work on NK-landscapes from Assignment 3.

In [1]:
```
# 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 s

import scipy.stats # for finding statistical significance

import time
```

Our NK fitness landscape function from Assignment 3.

In [2]:
```
class Landscape:
    """ N-K Fitness Landscape
    """


    def __init__(self, n=10, k=2):
        self.n = n # genome length
        self.k = k # number of other loci interacting with each gene
        self.gene_contribution_weight_matrix = np.random.rand(n,2**(k+1)) # for each gene,

    # find values of interacting loci
    def get_contributing_gene_values(self, genome, gene_num):
        contributing_gene_values = ""
        for i in range(self.k+1): # for each interacing loci (including the location of th
            contributing_gene_values += str(genome[(gene_num+i)%self.n]) # for simplicity
        return contributing_gene_values # return the string containing the values of all i

    # find the value of a partiuclar genome
    def get_fitness(self, genome):
        gene_values = np.zeros(self.n) # the value of each gene in the genome
        for gene_num in range(len(genome)): # for each gene
            contributing_gene_values = self.get_contributing_gene_values(genome, gene_num)
            gene_values[gene_num] = self.gene_contribution_weight_matrix[gene_num,int(cont
        return np.mean(gene_values) # define the fitness of the full genome as the average
```

Q1: Baseline implementation

Let's copy our usual `Individual` and `evolutionary_algorithm` setup from before. For simplicity in future questions, let's simplify our algorithm as much as possible, working with bit-string (as per the NK fitness function), mutation only (just flipping one bit) and no crossover, and simple truncation selection rather than tournament selection. Like last week, let's also record genotypic diverity over time.

In [17]:
```
class Individual:

    def __init__(self, fitness_function, bit_string_length):
        self.fitness_function = fitness_function
        self.genome = np.random.randint(2, size = bit_string_length)
        self.fitness = 0

    def eval_fitness(self):
        self.fitness = self.fitness_function(self.genome)
```

In [21]:
```
def evolutionary_algorithm(fitness_function=None, total_generations=100, num_parents=10, r
    """ 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 Lanc
        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 cour
        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
        crossover (bool): whether to perform crossover when generating children

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

    # initialize record keeping
    solution = None # best genome so far
    solution_fitness = -99999 # fitness of best genome so far
    fitness_over_time = np.zeros(total_generations)
    solutions_over_time = np.zeros((total_generations,bit_string_length))
    diversity_over_time = np.zeros(total_generations)

    # the initialization proceedure
    population = [] # keep population of individuals in a list
    for i in range(num_parents): # only create parents for initialization (the mu in mu+la
        population.append(Individual(fitness_function,bit_string_length)) # generate new i

    # get population fitness
    for i in range(len(population)):
        population[i].eval_fitness() # evaluate the fitness of each parent

    for generation_num in range(total_generations): # repeat

        # the modification procedure
        new_children = [] # keep children separate for now (lambda in mu+lambda)
        while len(new_children) < num_children:

            # inheretance
            [parent1, parent2] = np.random.choice(population, size=2) # pick 2 random pare
            child1 = copy.deepcopy(parent1) # initialize children as perfect copies of the
            child2 = copy.deepcopy(parent2)

            # crossover
            if crossover:
                [crossover_point1, crossover_point2] = sorted(np.random.randint(0,bit_stri
                child1.genome[crossover_point1:crossover_point2+1] = parent2.genome[crossc
                child2.genome[crossover_point1:crossover_point2+1] = parent1.genome[crossc

            # mutation
            for this_child in [child1,child2]:
                elements_to_mutate = set()
                while len(elements_to_mutate)<num_elements_to_mutate:
```

```
                elements_to_mutate.add(np.random.randint(bit_string_length)) # random
            for this_element_to_mutate in elements_to_mutate:
                this_child.genome[this_element_to_mutate] = (this_child.genome[this_el

        new_children.extend((child1,child2)) # add children to the new_children list

    # the assessement procedure
    for i in range(len(new_children)):
        new_children[i].eval_fitness() # assign fitness to each child

    # selection procedure
    population += new_children # combine parents with new children (the + in mu+lambda
    population = sorted(population, key=lambda individual: individual.fitness, reverse
    population = population[:num_parents] # perform truncation selection (keep just to

    # record keeping
    if population[0].fitness > solution_fitness: # if the new parent is the best found
        solution = population[0].genome                    # update best solution records
        solution_fitness = population[0].fitness
        solution_generation = generation_num
    fitness_over_time[generation_num] = solution_fitness # record the fitness of the c

    solutions_over_time[generation_num,:] = solution

    genome_list = np.array([individual.genome for individual in population])
    diversity = np.mean(genome_list.std(axis=0))
    diversity_over_time[generation_num] = diversity

return fitness_over_time, solutions_over_time, diversity_over_time
```

Initialize recordkeeping

In [22]:
```
experiment_results = {}
solutions_results = {}
diversity_results = {}
```

Q1b: Baseline Results

Let's pull all the pieces together and run `20` repitions of `100` generations of a population with `20` parents and `20` children. Let's use a NK-landscape with a bitstring length (`N`) of `15` and a highly rugged landscape of `K = 14` . (My repitions take about 1.5 second each)

In [23]:
```
num_runs = 20
total_generations = 100
num_elements_to_mutate = 1
bit_string_length = 15
num_parents = 20
num_children = 20
crossover = False

n = bit_string_length
k = bit_string_length - 1

for run_name in ["baseline"]:
    experiment_results[run_name] = np.zeros((num_runs, total_generations))
    solutions_results[run_name] = np.zeros((num_runs, total_generations, bit_string_length
    diversity_results[run_name] = np.zeros((num_runs, total_generations))
    for run_num in range(num_runs):
        landscape = Landscape(n=n, k=k)
        start_time = time.time()
        fitness_over_time, solutions_over_time, diversity_over_time = evolutionary_algorit
        experiment_results[run_name][run_num] = fitness_over_time
        solutions_results[run_name][run_num] = solutions_over_time
        diversity_results[run_name][run_num] = diversity_over_time
        print(run_name, run_num, time.time()-start_time, fitness_over_time[-1])
```

```
baseline 0 1.659430742263794 0.6767779520201855
baseline 1 1.583864688873291 0.7161287625556373
baseline 2 1.6899564266204834 0.700026381393952
baseline 3 1.7079722881317139 0.7079484878587146
baseline 4 1.5838651657104492 0.6699646960845226
baseline 5 1.7049691677093506 0.7200079976993252
baseline 6 1.7019665241241455 0.7101963764300009
baseline 7 1.5868678092956543 0.694609707658980 9
baseline 8 1.6979625225067139 0.704431511890242
baseline 9 1.6964619159698486 0.6783263973857384
baseline 10 1.5998785495758057 0.7294884539079637
baseline 11 1.6984636783599854 0.7050160104484637
baseline 12 1.6969623565673828 0.7186202467960006
baseline 13 1.5843653678894043 0.7633920152661947
baseline 14 1.7019667625427246 0.746549930052056
baseline 15 1.714477777481079 0.6995571452514386
baseline 16 1.572355031967163 0.7060247021150361
baseline 17 1.711975336074829 0.711666380087094
baseline 18 1.6929593086242676 0.7128103804818439
baseline 19 1.6133909225463867 0.7005605558689526
```

Q1c: Plotting

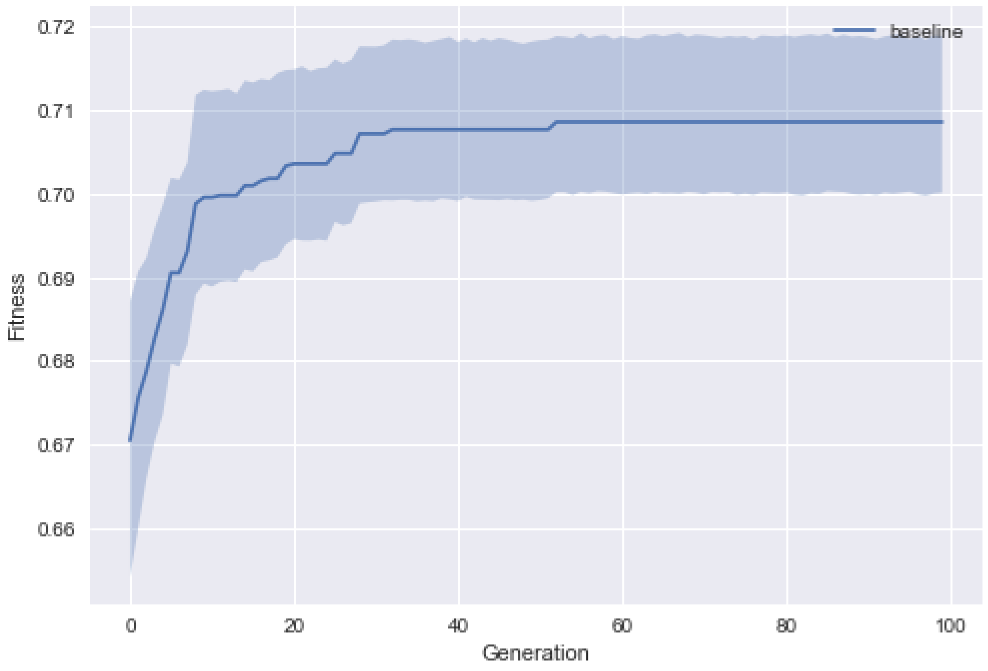
Please plot both the fitness over time and diversity over time of this run.

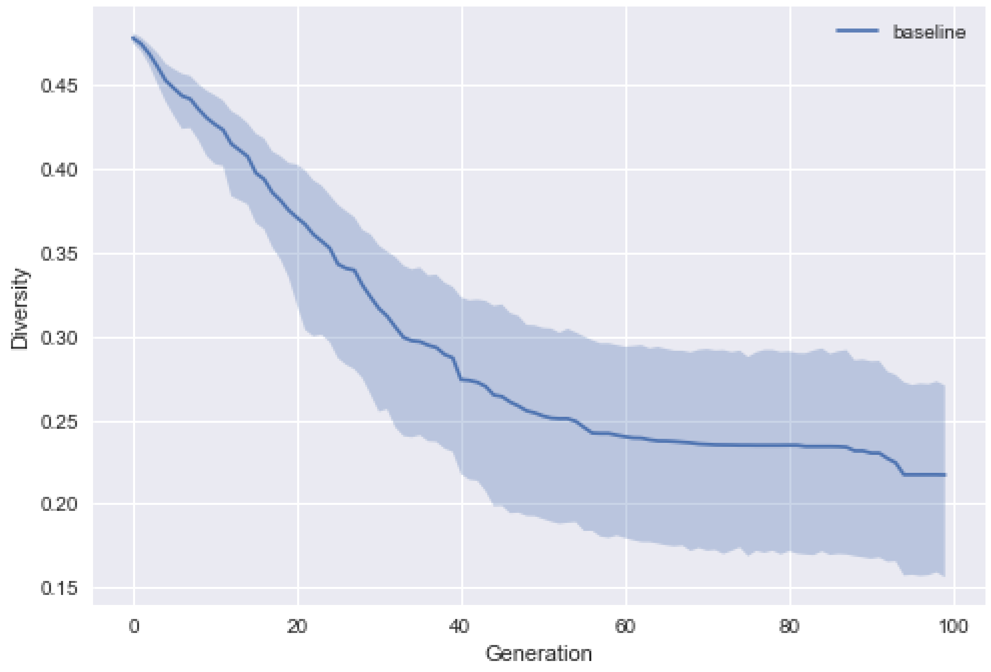
In [24]:

```
def plot_mean_and_bootstrapped_ci_over_time(input_data = None, name = "change me", x_label
    """

    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
    """

    fig, ax = plt.subplots() # generate figure and axes

    if isinstance(name, str): name = [name]; input_data = [input_data]

    # for this_input_data, this_name in zip(input_data, name):
    for this_name in name:
        print("plotting",this_name)
        this_input_data = input_data[this_name]
        total_generations = this_input_data.shape[1]

        if plot_bootstrap:
            boostrap_ci_generation_found = np.zeros((2,total_generations))
            for this_gen in range(total_generations):
                if this_gen%10==0: print(this_gen)
                boostrap_ci_generation_found[:,this_gen] = bootstrap.ci(this_input_data[:,


        ax.plot(np.arange(total_generations), np.mean(this_input_data,axis=0), label = thi
        if plot_bootstrap:
            ax.fill_between(np.arange(total_generations), boostrap_ci_generation_found[0,:
        ax.set_xlabel(x_label) # add axes labels
        ax.set_ylabel(y_label)
        if y_limit: ax.set_ylim(y_limit[0],y_limit[1])
        plt.legend(loc='best'); # add legend
```

In [25]:
```
# plotting
plot_mean_and_bootstrapped_ci_over_time(input_data = experiment_results, name = ["baseline
plot_mean_and_bootstrapped_ci_over_time(input_data = diversity_results, name = ["baseline'
```

```
plotting baseline
0
10
20
30
40
50
60
70
80
90
plotting baseline
0
10
20
30
40
50
60
70
80
90
```





Q2: Analysis

What do your results look like? In what ways, if any, do you expect that they'll change if we search for novelty instead of fitness? Why?

The results look like fitness goes up over time and diversity goes down over time. I expect that this will change with novelty in that the diversity will stay bounce around but stay within a higher range than the final diversity here. I also expect for the fitness to climb more slowly but steadily over time as the discovery of better solutions are found.

Q3: Implementing Novelty

Let's implement novelty search! First, modify your `Individual` class to record `novelty` as an attribute (in additon to `fitness`).

Hint: As ususal, you may want to skip ahead to the modification of your `evolutionary_algorithm` function in `Q4` and come back to fill in the helper functions in `Q3` once you have a better idea of when they'll be used (they just appear first to establish defniitions in case you `Restart and Run All`).

In [69]:
```python
class Individual:

    def __init__(self, fitness_function, bit_string_length):
        self.fitness_function = fitness_function
        self.novelty_function = get_novelty
        self.genome = np.random.randint(2, size = bit_string_length)
        self.fitness = 0
        self.novelty = 0

    def eval_fitness(self):
        self.fitness = self.fitness_function(self.genome)

    def eval_novelty(self, archive, k):
        if self.novelty == 0:
            self.novelty = self.novelty_function(archive, self, k)
```

Q3b: Caclulate Novelty

Let's define the novelty of a solution to be the average hamming/euclidean distance (i.e. number of differing bits) between the closest `k` genomes to it within an archive of prior solutions. It may be helpful to define a helper function to calculate this quanity.

In [108…

```python
# Function to find the cross over point (the point before which elements are
# smaller than or equal to x and after which greater than x)
def findCrossOver(arr, low, high, x) :

    # Base cases
    if (arr[high] <= x) : # x is greater than all
        return high

    if (arr[low] > x) : # x is smaller than all
        return low

    # Find the middle point
    mid = (low + high) // 2 # low + (high - low)// 2

    # If x is same as middle element, then return mid
    if (arr[mid] <= x and arr[mid + 1] > x) :
        return mid

    # If x is greater than arr[mid], then either arr[mid + 1] is ceiling of x
    # or ceiling lies in arr[mid+1...high]
    if(arr[mid] < x) :
        return findCrossOver(arr, mid + 1, high, x)

    return findCrossOver(arr, low, mid - 1, x)

# This function prints k closest elements to x in arr[]. n is the number of elements in a
def getKclosest(solutions, x, k, n) :

    arr = [s.fitness for s in solutions]
    neighbors = []

    # Find the crossover point
    l = findCrossOver(arr, 0, n - 1, x)
    r = l + 1 # Right index to search
    count = 0 # To keep track of count of elements already printed

    # If x is present in arr[], then reduce left index. Assumption: all elements
    # in arr[] are distinct
    if (arr[l] == x) :
        l -= 1

    # Compare elements on left and right of crossover point to find the k closest elements
    while (l >= 0 and r < n and count < k) :

        if (x - arr[l] < arr[r] - x) :
            neighbors.append(solutions[l])
#             print(arr[l], end = " ")
            l -= 1
        else :
            neighbors.append(solutions[r])
#             print(arr[r], end = " ")
            r += 1
        count += 1

    # If there are no more elements on right side, then print left elements
    while (count < k and l >= 0) :
        neighbors.append(solutions[l])
#         print(arr[l], end = " ")
        l -= 1
        count += 1
    # If there are no more elements on left side, then print right elements
    while (count < k and r < n) :
        neighbors.append(solutions[r])
#         print(arr[r], end = " ")
        r += 1
        count += 1

    return neighbors

def hamming_distance(s1, s2):
    """Calculate the Hamming distance between two bit strings"""
    return sum(c1 != c2 for c1, c2 in zip(s1, s2))


def get_novelty(solution_archive, individual, k):

    # get the k closest neighbors from solution_archive according to fitness
    closest = getKclosest(solution_archive, individual.fitness, k, n=len(solution_archive)
    novelty = 0
    novelty = sum([hamming_distance(individual.genome, c.genome) for c in closest])

    return novelty
```

Q3c: Selecting for Novelty

Please modify your evolutionary algorithm code to select (again, using the truncation selection as above) for the most novel solutions in our population (according to the novelty metric defined above), regardless of their fitness.

In order to keep down the cost of computing the distance between a new genome and all those that have previously existed, let's also set a finite size to our novelty archive (as a parameter we can pass to the algorithm). When we trying to add new genomes to the novelty archive, only add the the new individual if it has a higher novelty score than an individual already in the novelty archive (and remove that prior individual from the archive to keeep the archive size the same). Let's also say that the novelty of an individual will be its novelty score when first being considered for additon to the archive (i.e. we do not have to re-calculate it in the future as the makeup of the archive changes).

It may also be helpful to build a helper function for (potentially) updating the archive with a new individual.

Feel free to use the indidivuals in the current generation for your archive calculation or not, whichever is more convenient for your implementation.

Hint: If you've sorted the population by novelty for selection, don't forget that the population will no longer be in order of fitness when you go to record the fitness of most fit indidivual for record keeping!

In [109…

```
def update_archive(solution_archive, individual, max_archive_length):
    if len(solution_archive) == max_archive_length:
        if individual.novelty < sorted(solution_archive, key=lambda i: i.novelty)[0].novel
            return
        else:
            sorted(solution_archive, key=lambda i: i.novelty)[0] = copy.deepcopy(individua
    else:
        solution_archive.append(copy.deepcopy(individual))

    return solution_archive
```

In [110…

```
def evolutionary_algorithm(fitness_function=None, total_generations=100, num_parents=10, n
    """ 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 Lanc
        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 coun
        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
        crossover (bool): whether to perform crossover when generating children

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

    # initialize record keeping
    solution = None # best genome so far
    solution_fitness = -99999 # fitness of best genome so far
    fitness_over_time = np.zeros(total_generations)
    solutions_over_time = np.zeros((total_generations,bit_string_length))
    diversity_over_time = np.zeros(total_generations)
    solution_archive = []
    max_archive_length = 100

    # the initialization proceedure
    population = [] # keep population of individuals in a list
    for i in range(num_parents): # only create parents for initialization (the mu in mu+la
        population.append(Individual(fitness_function,bit_string_length)) # generate new i

    # get population fitness
    for i in range(len(population)):
        population[i].eval_fitness() # evaluate the fitness of each parent
        if len(solution_archive) < max_archive_length:
            solution_archive.append(population[i])
        else:
            population[i].eval_novelty(solution_archive, novelty_k)
            solution_archive = update_archive(solution_archive, population[i], max_archive



    for generation_num in range(total_generations): # repeat

        # the modification procedure
        new_children = [] # keep children separate for now (lambda in mu+lambda)
        while len(new_children) < num_children:

            # inheretance
            [parent1, parent2] = np.random.choice(population, size=2) # pick 2 random pare
            child1 = copy.deepcopy(parent1) # initialize children as perfect copies of the
            child2 = copy.deepcopy(parent2)

            # crossover
            if crossover:
                [crossover_point1, crossover_point2] = sorted(np.random.randint(0,bit_stri
                child1.genome[crossover_point1:crossover_point2+1] = parent2.genome[crosso
                child2.genome[crossover_point1:crossover_point2+1] = parent1.genome[crosso

            # mutation
            for this_child in [child1,child2]:
                elements_to_mutate = set()
                while len(elements_to_mutate)<num_elements_to_mutate:
                    elements_to_mutate.add(np.random.randint(bit_string_length)) # random
                for this_element_to_mutate in elements_to_mutate:
                    this_child.genome[this_element_to_mutate] = (this_child.genome[this_el
            new_children.extend((child1,child2)) # add children to the new_children list

        # the assessement procedure
        for i in range(len(new_children)):
            new_children[i].eval_fitness() # assign fitness to each child
            if len(solution_archive) < max_archive_length:
                solution_archive.append(new_children[i])
            else:
                new_children[i].eval_novelty(solution_archive, novelty_k)
                solution_archive = update_archive(solution_archive, new_children[i], max_a

        # selection procedure
        population += new_children # combine parents with new children (the + in mu+lambda
        for i in range(len(population)):
            population[i].eval_novelty(solution_archive, novelty_k)
        population = sorted(population, key=lambda individual: individual.novelty, reverse
        population = population[:num_parents] # perform truncation selection (keep just to

        # record keeping
        population = sorted(population, key=lambda individual: individual.fitness, reverse
        if population[0].fitness > solution_fitness: # if the new parent is the best found
            solution = population[0].genome                  # update best solution records
            solution_fitness = population[0].fitness
            solution_generation = generation_num
        fitness_over_time[generation_num] = solution_fitness # record the fitness of the c

        solutions_over_time[generation_num,:] = solution

        genome_list = np.array([individual.genome for individual in population])
        diversity = np.mean(genome_list.std(axis=0))
        diversity_over_time[generation_num] = diversity

    return fitness_over_time, solutions_over_time, diversity_over_time
```

Q4: Run

Run novelty search with the same hyperparameter settings as above, and with a `novelty_k` value of `5` nearest neighbors for calculating the novelty metrics, from an archive of the `100` most novel individuals found thus far. (My runs take about 3 seconds for each repitition)

In [111…

```
num_runs = 20
total_generations = 100
num_elements_to_mutate = 1
bit_string_length = 15
num_parents = 20
num_children = 20
novelty_k = 5
crossover = False

n = bit_string_length
k = bit_string_length - 1

for run_name in ["novelty"]:
    experiment_results[run_name] = np.zeros((num_runs, total_generations))
    solutions_results[run_name] = np.zeros((num_runs, total_generations, bit_string_length
    diversity_results[run_name] = np.zeros((num_runs, total_generations))
    for run_num in range(num_runs):
        landscape = Landscape(n=n, k=k)
        start_time = time.time()
        fitness_over_time, solutions_over_time, diversity_over_time = evolutionary_algorit
        experiment_results[run_name][run_num] = fitness_over_time
        solutions_results[run_name][run_num] = solutions_over_time
        diversity_results[run_name][run_num] = diversity_over_time
        print(run_name, run_num, time.time()-start_time, fitness_over_time[-1])
```

```
novelty 0 3.2833290100097656 0.6631861935917315
novelty 1 3.0456249713897705 0.6243479668510575
novelty 2 3.1507158279418945 0.6753459163888358
novelty 3 3.0916643142700195 0.6718015375579586
novelty 4 3.086660146713257 0.6349129344029324
novelty 5 3.096168279647827 0.6445048647337647
novelty 6 3.152717351913452 0.6271033878943465
novelty 7 3.1422080993652344 0.6115307681087006
novelty 8 3.212768793106079 0.6380103237488798
novelty 9 3.133200168609619 0.6692972831480954
novelty 10 3.1166858673095703 0.6583704577765559
novelty 11 3.0546326637268066 0.6650636155794061
novelty 12 3.1417078971862793 0.6546464642760302
novelty 13 3.136202812194824 0.6937105220197403
novelty 14 3.092665433883667 0.6910873723498978
novelty 15 3.0941665172576904 0.7183962402697087
novelty 16 3.2508020401000977 0.6588626586028172
novelty 17 3.1722335815429688 0.6705395149982715
novelty 18 3.1231915950775146 0.698239983695902
novelty 19 3.0851588249206543 0.6681330818592334
```

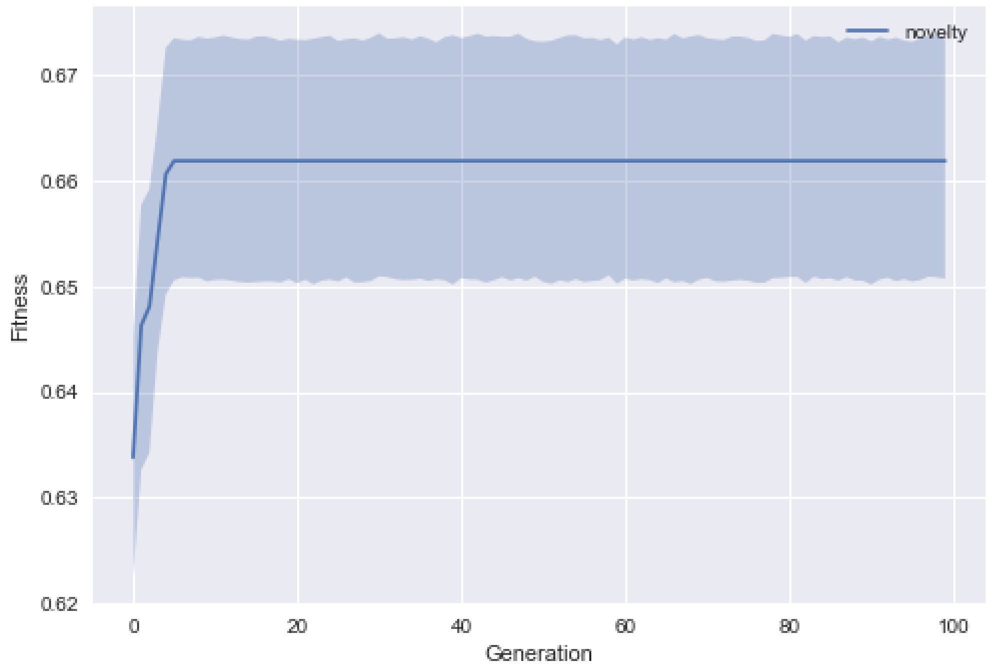
Q4b: Plot

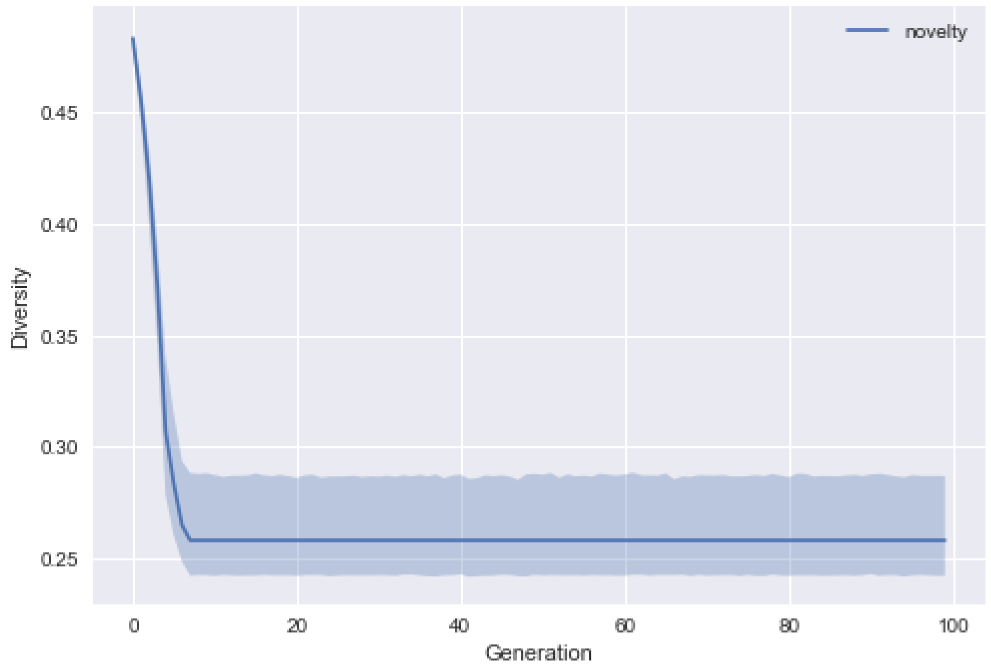
Please visualize the fitness and diversity over time for novelty search vs. fitness-based search

In [112…

```
# plotting
plot_mean_and_bootstrapped_ci_over_time(input_data = experiment_results, name = ["novelty'
plot_mean_and_bootstrapped_ci_over_time(input_data = diversity_results, name = ["novelty"]
```

```
plotting novelty
0
10
20
30
40
50
60
70
80
90
plotting novelty
0
10
20
30
40
50
60
70
80
90
```





Q5: Analysis

How does novelty search perform in this domain? Is it what you expected? If no, why might that be the case?

Novelty seems to perform very poorly. This is not what I was expecting, but upon thinking more deeply about this particular situation, it might start to make sense. We are calculating novelty as the hamming distance between the genomes in the archive with the closest fitness. This is a phenotypic type measurement of novelty, whereas our diversity calculation is completed genotypically. This still doesn't explain everything since we can see even our fitness values begin to converge fairly quickly. I believe this is because we only calculate novelty the first time an individual is assessed. We do actually maintain a higher level of diversity in the end than the fitness based evolution, but our fitness scores do not do as well as I imagined. I also wonder if this has something to do with the ruggedness of the landscape and the calculation of novelty only being the 5 nearest fitness values.

Q6: Larger Neighborhoods

How might you expect the result to change if you were to use a larger neighborhood (`novelty_k` value) for calculating a soluton's novelty.

Let's try it! Please run and plot the same settings as above, but with a `novelty_k` of `100` (i.e. using the full archive for novelty calcuation).

In [113…

```
num_runs = 20
total_generations = 100
num_elements_to_mutate = 1
bit_string_length = 15
num_parents = 20
num_children = 20
novelty_k = 100
crossover=False

n = bit_string_length
k = bit_string_length-1

for run_name in ["novelty_100"]:
    experiment_results[run_name] = np.zeros((num_runs, total_generations))
    solutions_results[run_name] = np.zeros((num_runs, total_generations, bit_string_length
    diversity_results[run_name] = np.zeros((num_runs, total_generations))
    for run_num in range(num_runs):
        landscape = Landscape(n=n, k=k)
        start_time = time.time()
        fitness_over_time, solutions_over_time, diversity_over_time = evolutionary_algorit
        experiment_results[run_name][run_num] = fitness_over_time
        solutions_results[run_name][run_num] = solutions_over_time
        diversity_results[run_name][run_num] = diversity_over_time
        print(run_name, run_num, time.time()-start_time, fitness_over_time[-1])
```

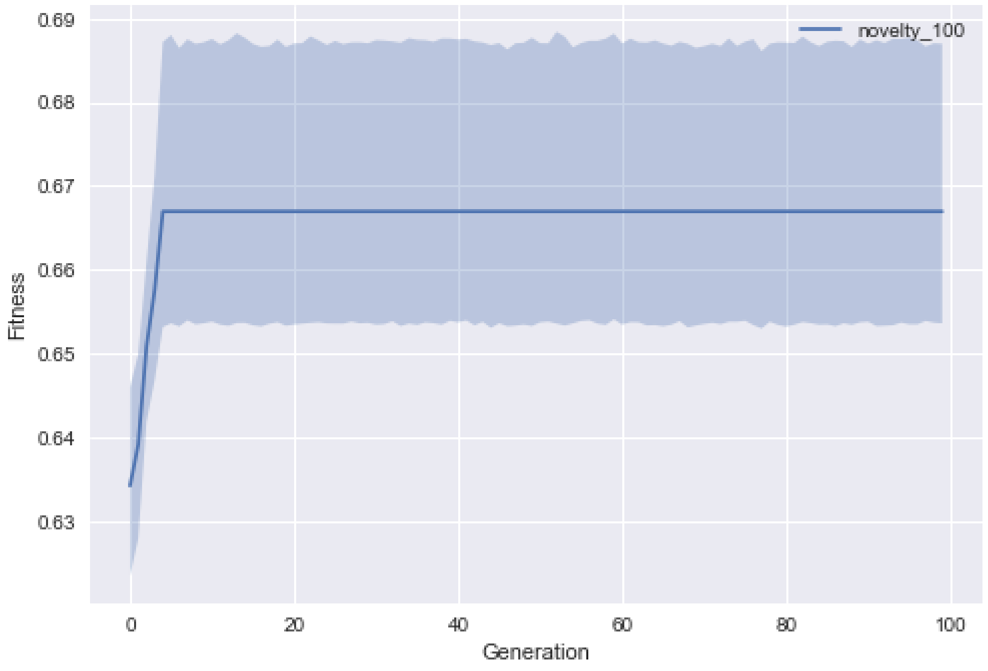
```
novelty_100 0 3.0711467266082764 0.6743234940789723
novelty_100 1 3.110680341720581 0.6971062391030275
novelty_100 2 3.127195358276367 0.6621278545089613
novelty_100 3 3.0906636714935303 0.6202840054727196
novelty_100 4 3.138204336166382 0.6505909957256649
novelty_100 5 3.174736738204956 0.6618290295557366
novelty_100 6 3.181241989135742 0.6422001551985107
novelty_100 7 3.1226911544799805 0.6433890993586838
novelty_100 8 3.144709825515747 0.701519218862463
novelty_100 9 3.127694845199585 0.6503057156420744
novelty_100 10 3.058135509490967 0.6382798955443749
novelty_100 11 3.1286962032318115 0.6764684939707392
novelty_100 12 3.120690107345581 0.6653501137770769
novelty_100 13 3.1927506923675537 0.6363231454578321
novelty_100 14 3.1332004070281982 0.6926361687289061
novelty_100 15 3.064140796661377 0.6674163625830875
novelty_100 16 3.093665838241577 0.625065588887792
novelty_100 17 3.0921647548675537 0.7740976042728948
novelty_100 18 3.1452107429504395 0.7317639348086847
novelty_100 19 3.0741496086120605 0.6280950694572478
```

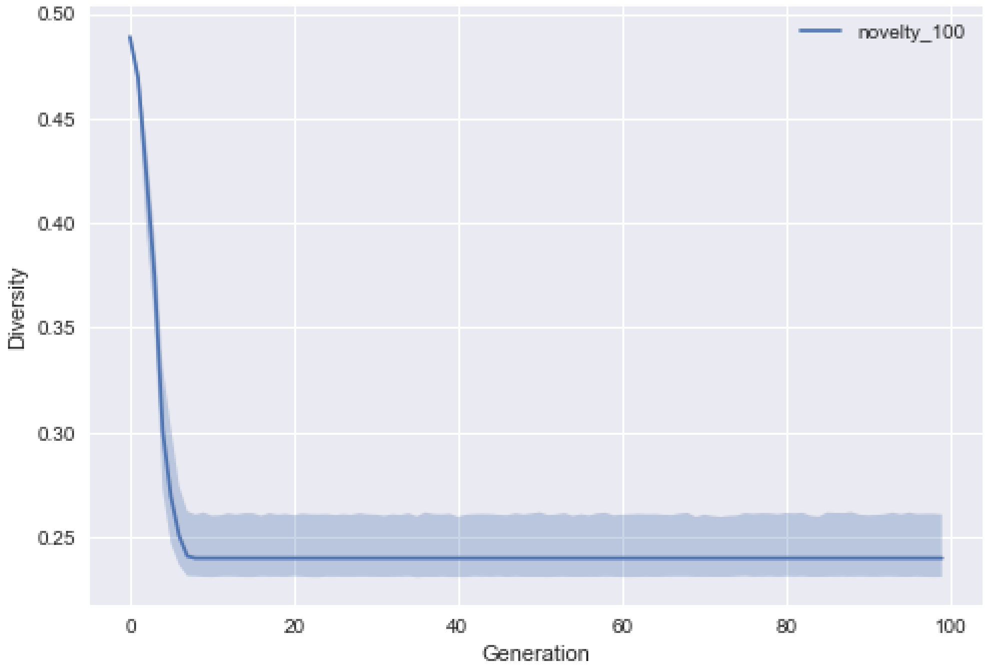
In [114…

```
# plotting
plot_mean_and_bootstrapped_ci_over_time(input_data = experiment_results, name = ["novelty_
plot_mean_and_bootstrapped_ci_over_time(input_data = diversity_results, name = ["novelty_
```

```
plotting novelty_100
0
10
20
30
40
50
```

```
60
70
80
90
plotting novelty_100
0
10
20
30
40
50
60
70
80
90
```





Q6b: Analysis

What happened? Did it work better or worse? Is this what you expected (and why)?

It looks like we end up with less diversity in the end with a slightly higher fitness in the end. This is

interesting as well. It doesn't seem to have changed much. It isn't what I expected. I would have expected higher number os diversity since we are taking novelty as the hamming distance between the individual's fitness and every other fitness in the solution archive instead of just 5.

Q7: Mixed Fitness and Novelty

As suggested in class, perhaps the best version of an evolutionary algorithm is not one that selects just for fitness or one that selects just for novelty, but one that considers both in an attempt to carefully tradeoff exploration and exploitation.

We may not be the most careful and inentional with our tradeoffs here, but let's start with perhaps the simplest combination of selecting for both novelty and fitness one could think of. Let's select some of the individuals in our population on the basis of novelty and some on the basis of fitness.

In particular, pleae define a new parameter `novelty_selection_prop` that defines what proportion of the parents for the next generation will be selected by novelty (and choose the most novel solutions to occupy that portion of the new generation) while the remainder of the new population (`1-novelty_selection_prop`) gets selected on the basis of fitness -- resulting in the same `num_parents` as before heading into the next generation.

In [115…

```
def evolutionary_algorithm(fitness_function=None, total_generations=100, num_parents=10, r
    """ 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 Land
        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 cour
        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
        crossover (bool): whether to perform crossover when generating children

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

    # initialize record keeping
    solution = None # best genome so far
    solution_fitness = -99999 # fitness of best genome so far
    fitness_over_time = np.zeros(total_generations)
    solutions_over_time = np.zeros((total_generations,bit_string_length))
    diversity_over_time = np.zeros(total_generations)
    solution_archive = []
    max_archive_length = 100

    # the initialization proceedure
    population = [] # keep population of individuals in a list
    for i in range(num_parents): # only create parents for initialization (the mu in mu+la
        population.append(Individual(fitness_function,bit_string_length)) # generate new i

    # get population fitness
    for i in range(len(population)):
        population[i].eval_fitness() # evaluate the fitness of each parent
        if len(solution_archive) < max_archive_length:
            solution_archive.append(population[i])
        else:
            population[i].eval_novelty(solution_archive, novelty_k)
            solution_archive = update_archive(solution_archive, population[i], max_archive
    for generation_num in range(total_generations): # repeat

        # the modification procedure
        new_children = [] # keep children separate for now (lambda in mu+lambda)
        while len(new_children) < num_children:

            # inheretance
            [parent1, parent2] = np.random.choice(population, size=2) # pick 2 random pare
            child1 = copy.deepcopy(parent1) # initialize children as perfect copies of the
            child2 = copy.deepcopy(parent2)

            # crossover
            if crossover:
                [crossover_point1, crossover_point2] = sorted(np.random.randint(0,bit_stri
                child1.genome[crossover_point1:crossover_point2+1] = parent2.genome[crosso
                child2.genome[crossover_point1:crossover_point2+1] = parent1.genome[crosso

            # mutation
            for this_child in [child1,child2]:
                elements_to_mutate = set()
                while len(elements_to_mutate)<num_elements_to_mutate:
                    elements_to_mutate.add(np.random.randint(bit_string_length)) # random.
                for this_element_to_mutate in elements_to_mutate:
                    this_child.genome[this_element_to_mutate] = (this_child.genome[this_el

            new_children.extend((child1,child2)) # add children to the new_children list

        # the assessement procedure
        for i in range(len(new_children)):
            new_children[i].eval_fitness() # assign fitness to each child
            if len(solution_archive) < max_archive_length:
                solution_archive.append(new_children[i])
            else:
                new_children[i].eval_novelty(solution_archive, novelty_k)
                solution_archive = update_archive(solution_archive, new_children[i], max_a

        # selection procedure
        population += new_children # combine parents with new children (the + in mu+lambda
        for i in range(len(population)):
            population[i].eval_novelty(solution_archive, novelty_k)
        new_population = sorted(population, key=lambda individual: individual.novelty, rev
        new_population = population[:num_parents*novelty_selection_prop] # perform truncat
        population = sorted(population, key=lambda individual: individual.fitness, reverse
        population = population[:num_parents*(1-novelty_selection_prop)] # perform truncat
        population.extend(new_population)

        # record keeping
        population = sorted(population, key=lambda individual: individual.fitness, reverse
        if population[0].fitness > solution_fitness: # if the new parent is the best found
            solution = population[0].genome                 # update best solution records
            solution_fitness = population[0].fitness
            solution_generation = generation_num
        fitness_over_time[generation_num] = solution_fitness # record the fitness of the c

        solutions_over_time[generation_num,:] = solution

        genome_list = np.array([individual.genome for individual in population])
        diversity = np.mean(genome_list.std(axis=0))
        diversity_over_time[generation_num] = diversity

    return fitness_over_time, solutions_over_time, diversity_over_time
```

Q8: Experimentation

Let's try running this mixed selection criteria for a 50/50 split between suvivors/parents for the next generation selected via novelty vs. fitness. Let's do this with our original novelty neighbhorhood of size 5, and all other parameters the same.

As usual, please plot fitness and diversity afterwards.

In [116…

```
num_runs = 20
total_generations = 100
num_elements_to_mutate = 1
bit_string_length = 15
num_parents = 20
num_children = 20
crossover=False

novelty_k = 5
novelty_selection_prop = 0.5
max_archive_length = 100

n = bit_string_length
k = bit_string_length-1

for run_name in ["mixed"]:
    experiment_results[run_name] = np.zeros((num_runs, total_generations))
    solutions_results[run_name] = np.zeros((num_runs, total_generations, bit_string_lengtl
    diversity_results[run_name] = np.zeros((num_runs, total_generations))
    for run_num in range(num_runs):
        landscape = Landscape(n=n, k=k)
        start_time = time.time()
        fitness_over_time, solutions_over_time, diversity_over_time = evolutionary_algorit
        experiment_results[run_name][run_num] = fitness_over_time
        solutions_results[run_name][run_num] = solutions_over_time
        diversity_results[run_name][run_num] = diversity_over_time
        print(run_name, run_num, time.time()-start_time, fitness_over_time[-1])
```

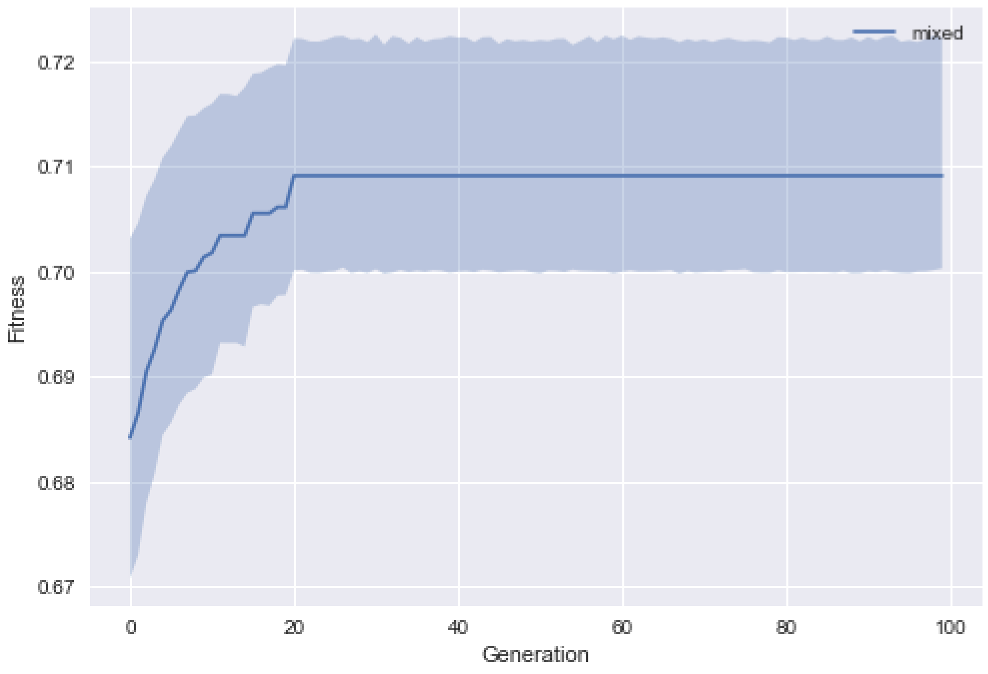
```
mixed 0 3.033113956451416 0.7089723773258447
mixed 1 3.087660789489746 0.6882902947721087
mixed 2 3.1051764488220215 0.7290663316080586
mixed 3 3.2683165073394775 0.7293028760645053
mixed 4 3.055133104324341 0.6954505315940167
mixed 5 3.1061770915985107 0.6922160204424405
mixed 6 3.067143678665161 0.7314006587336119
mixed 7 3.065141201019287 0.7023106352244834
mixed 8 3.101172685623169 0.6888131035358994
mixed 9 3.0541319847106934 0.7766807160646392
mixed 10 3.140706777572632 0.7069033135297254
mixed 11 3.090163230895996 0.6839305278807112
mixed 12 3.317859649658203 0.7093227769835185
mixed 13 3.1221909523010254 0.6780206784130453
mixed 14 3.078152656555176 0.6973473394158738
mixed 15 3.089662551879883 0.7166927126288233
mixed 16 3.0791537761688232 0.6782114187468163
mixed 17 3.1311984062194824 0.7171617478693236
mixed 18 3.047125816345215 0.7533112977376779
mixed 19 3.1281962394714355 0.6990935852382691
```

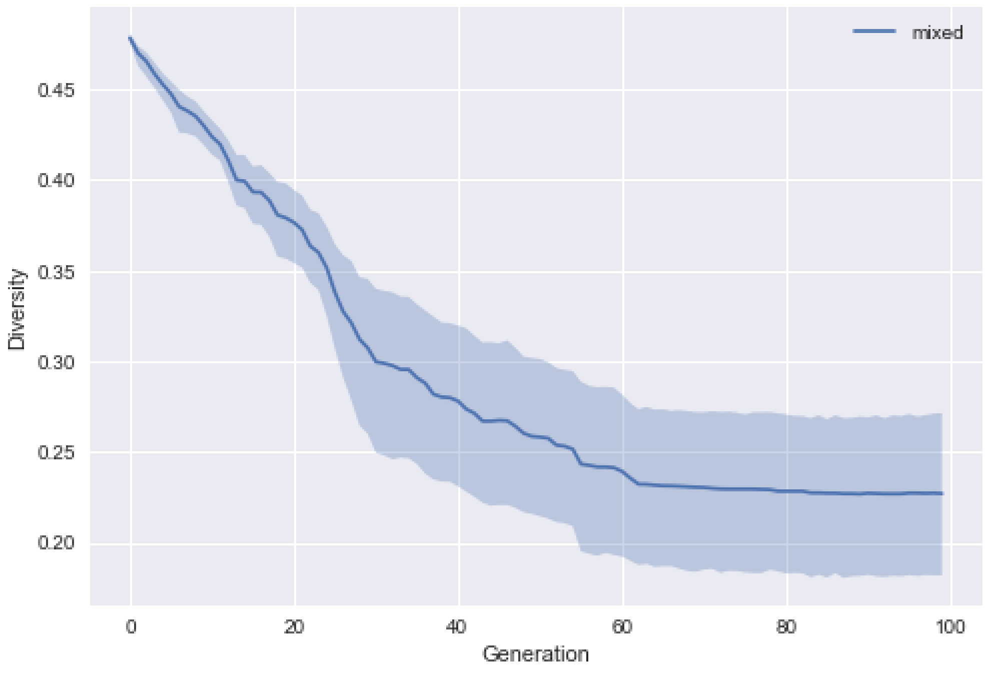
In [117…

```
# plotting
plot_mean_and_bootstrapped_ci_over_time(input_data = experiment_results, name = ["mixed"],
plot_mean_and_bootstrapped_ci_over_time(input_data = diversity_results, name = ["mixed"],
```

```
plotting mixed
```

```
0
10
20
30
40
50
60
70
80
90
plotting mixed
0
10
20
30
40
50
60
70
80
90
```





Q9: Analysis

What happened (to both fitness and diversity)? Are you surprised? Why would this be?

This looks very similar to the fitness based evolutionary algorithm. There is slightly more diversity and the standard deviation seems to be a bit wider in regards to fitness. The standard deviation for diversity actually seems smaller than the fitness-based approach. I am a bit surprised that this doesn't differ more significantly from the fitness-based evolutionary algorithm. I think that maybe it is similar because it injects the selection pressure back into the algorithm which narrows the exploration of novelty search.

Q10: Balancing Novelty and Fitness

Let's run this again with a different balance of novelty vs. fitness. Please run it with 90% of survivors selected via novelty and just 10% selected via fitness, and also vise versa with just 10% novelty and 90% fitness at each generation. Which do you expect to work better?

Judging from the previous charts, I think that the 90% fitness will work better than the 90% novelty since 100% novelty was not able to find very good solutions.

Q9b: Running and Visualization

Let's findout!

In [118…

```
num_runs = 20
total_generations = 100
num_elements_to_mutate = 1
bit_string_length = 15
num_parents = 20
num_children = 20
crossover=False

novelty_k = 5
novelty_selection_prop = 0
max_archive_length = 100

n = bit_string_length
k = bit_string_length-1

for run_name in ["10%_novelty", "90%_novelty"]:
    if run_name == "10%_novelty":
        novelty_selection_prop = 0.1
    else:
        novelty_selection_prop = 0.9
    experiment_results[run_name] = np.zeros((num_runs, total_generations))
    solutions_results[run_name] = np.zeros((num_runs, total_generations, bit_string_length
    diversity_results[run_name] = np.zeros((num_runs, total_generations))
    for run_num in range(num_runs):
        landscape = Landscape(n=n, k=k)
        start_time = time.time()
        fitness_over_time, solutions_over_time, diversity_over_time = evolutionary_algorit
        experiment_results[run_name][run_num] = fitness_over_time
        solutions_results[run_name][run_num] = solutions_over_time
        diversity_results[run_name][run_num] = diversity_over_time
        print(run_name, run_num, time.time()-start_time, fitness_over_time[-1])
```

```
10%_novelty 0 3.1151845455169678 0.6894870715672116
10%_novelty 1 3.1196885108947754 0.7357852666547837
10%_novelty 2 3.0921640396118164 0.6770666782694363
10%_novelty 3 3.121690511703491 0.7071176877407189
10%_novelty 4 3.1046760082244873 0.6851527274662526
```

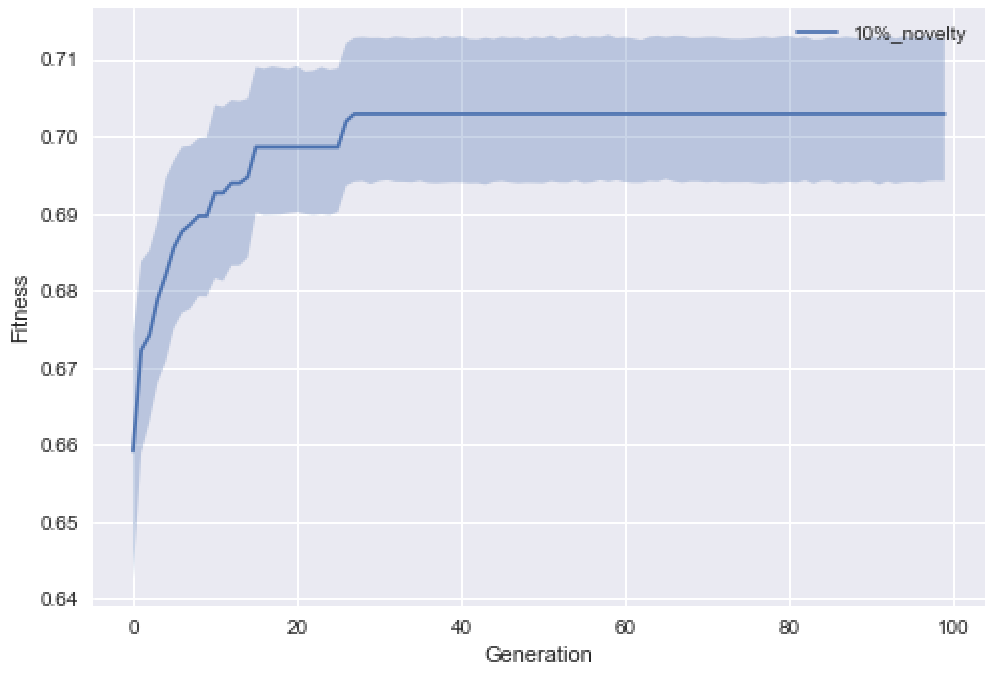
```
10%_novelty 5 3.089662551879883 0.6807168129950528
10%_novelty 6 3.095167398452759 0.7297566776821831
10%_novelty 7 3.070145845413208 0.6661318110327538
10%_novelty 8 3.2798268795013428 0.7219149424009129
10%_novelty 9 3.0666427612304688 0.7171957622311772
10%_novelty 10 3.0976691246032715 0.7314176463274311
10%_novelty 11 3.126194715499878 0.697577991677629
10%_novelty 12 3.0946669578552246 0.6852395447751036
10%_novelty 13 3.087661027908325 0.708466583729392
10%_novelty 14 3.147712469100952 0.7111105980055981
10%_novelty 15 3.0571346282958984 0.7462803900727127
10%_novelty 16 3.0401201248168945 0.6863895215044127
10%_novelty 17 3.1487133502960205 0.6882951227038542
10%_novelty 18 3.1652252674102783 0.6920564444681275
10%_novelty 19 3.08115553855896 0.7006918179724716
90%_novelty 0 3.0576348304748535 0.6924272717229201
90%_novelty 1 3.1196885108947754 0.7414516626696721
90%_novelty 2 3.07014536857605 0.6964082640377004
90%_novelty 3 3.1196885108947754 0.6654673575904755
90%_novelty 4 3.0791537761688232 0.7352661019479675
90%_novelty 5 3.131699323654175 0.741495147758817
90%_novelty 6 3.283829927444458 0.6851671702463367
90%_novelty 7 3.110180616378784 0.7376120854815856
90%_novelty 8 3.1251935958862305 0.6955319279477509
90%_novelty 9 3.075150489807129 0.7292061767822641
90%_novelty 10 3.041120767593384 0.7799632868136512
90%_novelty 11 3.084157943725586 0.6857956089125519
90%_novelty 12 3.1477131843566895 0.7092089275937151
90%_novelty 13 3.124192476272583 0.7101952482093423
90%_novelty 14 3.321361780166626 0.7194131075173155
90%_novelty 15 3.1762373447418213 0.7268280152423349
90%_novelty 16 3.067643642425537 0.7467430307005604
90%_novelty 17 3.1316988468170166 0.7169802839151314
90%_novelty 18 3.0856592655181885 0.7549268832728304
90%_novelty 19 3.1056768894195557 0.7170381983795552
```

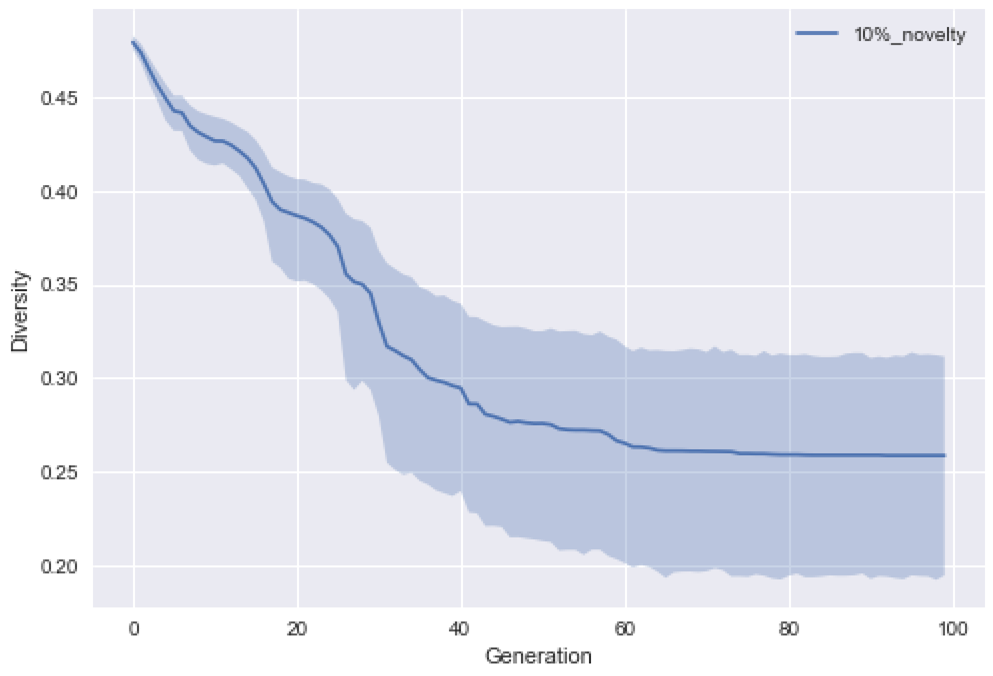
In [120…

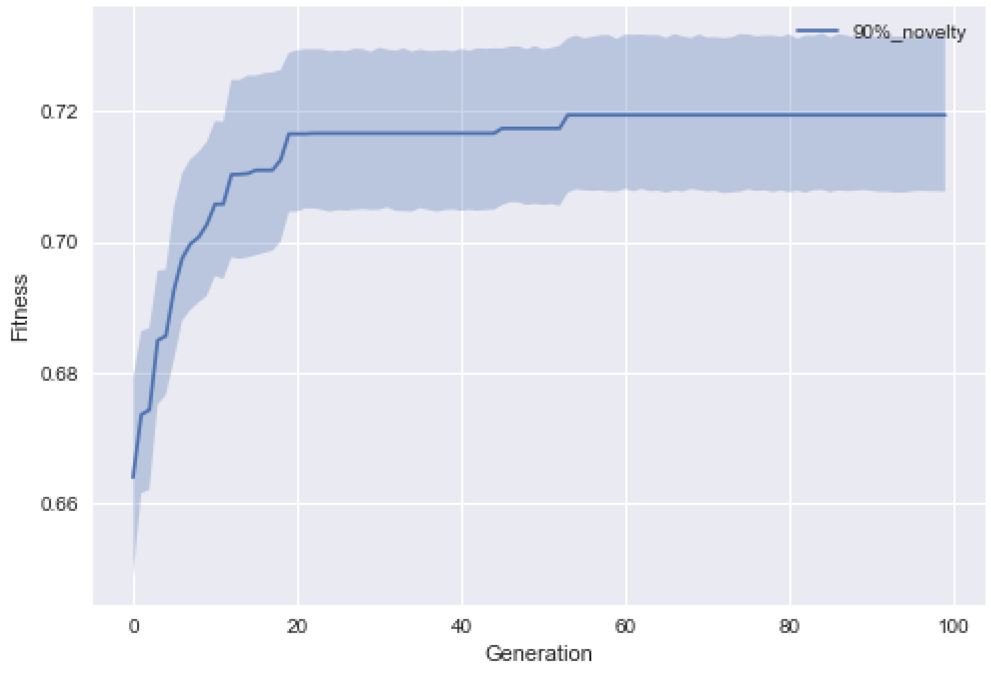
```python
# plotting
plot_mean_and_bootstrapped_ci_over_time(input_data = experiment_results, name = ["10%_nove
plot_mean_and_bootstrapped_ci_over_time(input_data = diversity_results, name = ["10%_novel
plot_mean_and_bootstrapped_ci_over_time(input_data = experiment_results, name = ["90%_nove
plot_mean_and_bootstrapped_ci_over_time(input_data = diversity_results, name = ["90%_novel
```

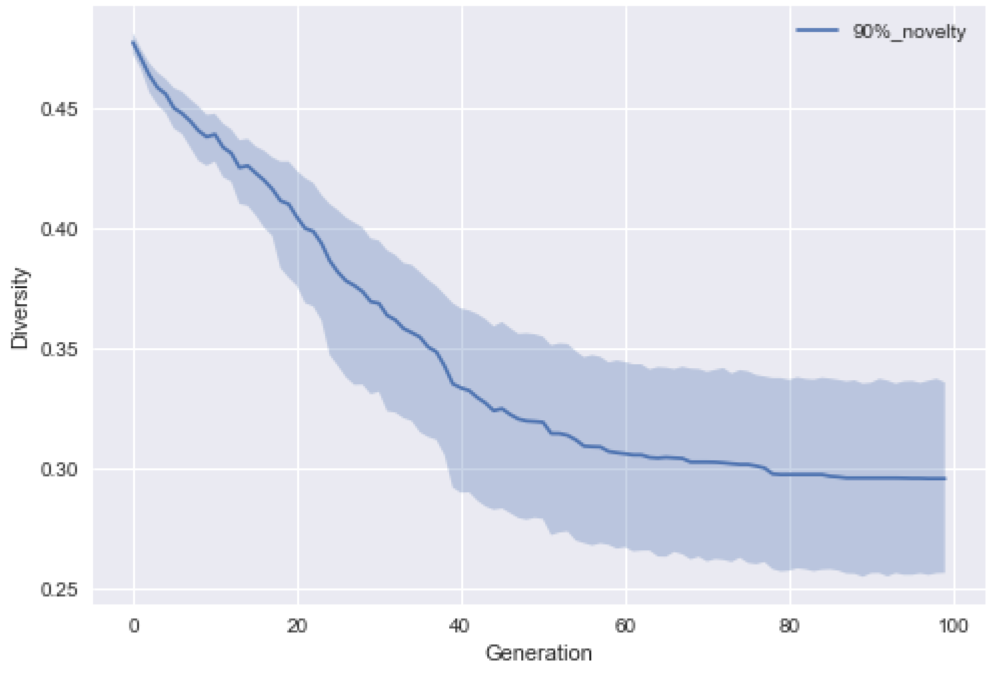
```
plotting 10%_novelty
0
10
20
30
40
50
60
70
80
90
plotting 10%_novelty
0
10
20
30
40
50
60
70
80
90
plotting 90%_novelty
```

```
0
10
20
30
40
50
60
70
80
90
plotting 90%_novelty
0
10
20
30
40
50
60
70
80
90
```









Q10: Analysis

Did the experiment turn out the way you thought it would? Why or why not? What does this imply about the use of novelty vs. fitness in expoitation vs. exploration? Do the diveristy plots support this idea?

It did not go as I thought it might. The 90% novelty (10% fitness), actually did quite well and better than any of the experiments so far. It had higher levels of diversity, even if they were declining (slowly), and it also had higher fitness than any of the previous cases. This implies that novelty can help explore, but it might also need some slight pressure to make decent progress on the specific metric for which we care about. It does very well at exploring the landscape and then exploiting just enough to do better than the rest of the cases. The plots support this implication.

Q11: The Effect of Ruggedness

How much do you think the conclusions you came to above are the result of the particular (maximally rugged) fitness landscape we experiemented with? What would happen if we used a much smoother landscale (e.g. a NK landscape with `K=0`)

I think it is very dependent on the landscape. If we used something more smooth, a selection process with more fitness bias might do better. I think that the novelty bias above does well because of the ruggedness of the landscape.

Q12: Experiementation

Let's find out! Please pick your best ratio of novelty vs. fitness selection, and compare it to purely fitness and purely novelty selection on a NK landscale with `K=0` . Please plot your results.

In [121…

```
num_runs = 20
total_generations = 100
num_elements_to_mutate = 1
bit_string_length = 15
num_parents = 20
num_children = 20

novelty_k = 5
novelty_selection_prop = 0
max_archive_length = 100

n = bit_string_length
k = 0

for run_name in ["novelty_k0", "90%_novelty_k0", "fitness_k0"]:
    if run_name == "90%_novelty":
        novelty_selection_prop = 0.9
    elif run_name == "novelty_k0":
        novelty_selection_prop = 1
    else:
        novelty_selection_prop = 0
    experiment_results[run_name] = np.zeros((num_runs, total_generations))
    solutions_results[run_name] = np.zeros((num_runs, total_generations, bit_string_length
    diversity_results[run_name] = np.zeros((num_runs, total_generations))
    for run_num in range(num_runs):
        landscape = Landscape(n=n, k=k)
        start_time = time.time()
        fitness_over_time, solutions_over_time, diversity_over_time = evolutionary_algorit
        experiment_results[run_name][run_num] = fitness_over_time
        solutions_results[run_name][run_num] = solutions_over_time
        diversity_results[run_name][run_num] = diversity_over_time
        print(run_name, run_num, time.time()-start_time, fitness_over_time[-1])
```

```
novelty_k0 0 0.2627263069152832 0.6014584990702649
novelty_k0 1 0.2682311534881592 0.575683148663465
novelty_k0 2 0.2672302722930908 0.7399304243112534
novelty_k0 3 0.2652285099029541 0.6104334015541241
novelty_k0 4 0.2707333564758301 0.6862516069998084
novelty_k0 5 0.26873087882995605 0.6710380249266842
novelty_k0 6 0.28324413299560547 0.6713132267287522
novelty_k0 7 0.26873159408569336 0.503346366353624
novelty_k0 8 0.27623820304870605 0.46887128038233694
novelty_k0 9 0.268230676651001 0.6335070733765124
novelty_k0 10 0.27173352241516113 0.7579577410841826
novelty_k0 11 0.2647287845611572 0.624535967361945
novelty_k0 12 0.265228271484375 0.6176280677170889
novelty_k0 13 0.26172566413879395 0.5966239775574865
novelty_k0 14 0.2632267475128174 0.6494677564826966
novelty_k0 15 0.270233154296875 0.5999618526425845
novelty_k0 16 0.2697319984436035 0.7462981716780727
novelty_k0 17 0.27373576164245605 0.6045802166908546
novelty_k0 18 0.2672300338745117 0.5780302748312788
novelty_k0 19 0.267730712890625 0.7589074172533059
```

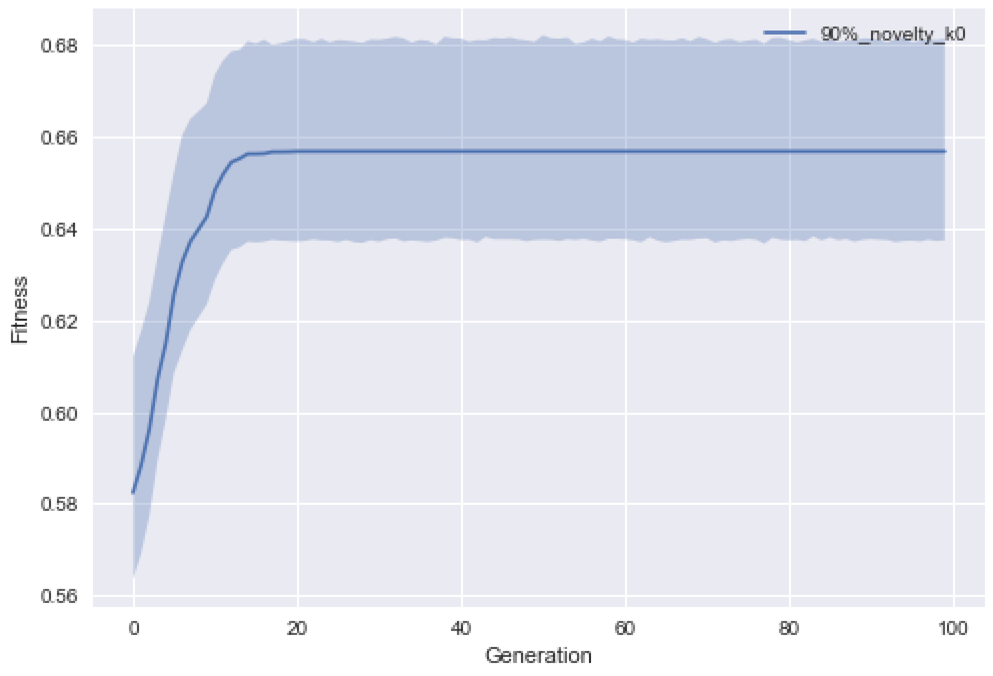
```
90%_novelty_k0 0 0.26372742652893066 0.6365272463550737
90%_novelty_k0 1 0.26122522354125977 0.6060838532554269
90%_novelty_k0 2 0.27123379707336426 0.6666432014171756
90%_novelty_k0 3 0.2627265453338623 0.7398103356169545
90%_novelty_k0 4 0.26122522354125977 0.6453641784442831
90%_novelty_k0 5 0.26873135566711426 0.5901526732218668
90%_novelty_k0 6 0.26372766494750977 0.5787859317609387
90%_novelty_k0 7 0.2682309150695801 0.718513517842045
90%_novelty_k0 8 0.26622939109802246 0.6495364718003127
90%_novelty_k0 9 0.2652285099029541 0.649917375200396
90%_novelty_k0 10 0.2672300338745117 0.6169982579878346
90%_novelty_k0 11 0.270233154296875 0.6439765546726469
90%_novelty_k0 12 0.26622962951660156 0.6391873225485996
90%_novelty_k0 13 0.262225866317749 0.6245889667005508
90%_novelty_k0 14 0.27123379707336426 0.6538625266053198
90%_novelty_k0 15 0.26422810554504395 0.7371965708020677
90%_novelty_k0 16 0.27673816680908203 0.6843347701201666
90%_novelty_k0 17 0.2747361660003662 0.7680689840313144
90%_novelty_k0 18 0.2707333564758301 0.6198357917128413
90%_novelty_k0 19 0.2682313919067383 0.6682182001043315
fitness_k0 0 0.268230676651001 0.6766850357068499
fitness_k0 1 0.2707333564758301 0.691174865940314
fitness_k0 2 0.2652285099029541 0.640439153947989
fitness_k0 3 0.2652285099029541 0.680452836573591
fitness_k0 4 0.26372742652893066 0.6548319602536794
fitness_k0 5 0.2632267475128174 0.6591214159903239
fitness_k0 6 0.2672305107116699 0.7327434547899058
fitness_k0 7 0.26672983169555664 0.6944411823402735
fitness_k0 8 0.2652285099029541 0.6950422539312903
fitness_k0 9 0.26673007011413574 0.577077005315743
fitness_k0 10 0.26622915267944336 0.666368533415623
fitness_k0 11 0.27173423767089844 0.8080298657495905
fitness_k0 12 0.2682311534881592 0.5601666829952376
fitness_k0 13 0.27173399925231934 0.738708487127589
fitness_k0 14 0.267730712890625 0.6054433542307682
fitness_k0 15 0.2682313919067383 0.7730977996664147
fitness_k0 16 0.26672983169555664 0.5883656057065216
fitness_k0 17 0.25922346115112305 0.6282970971284086
fitness_k0 18 0.2647278308868408 0.5878874788169451
fitness_k0 19 0.264728307723999 0.6371102367199295
```

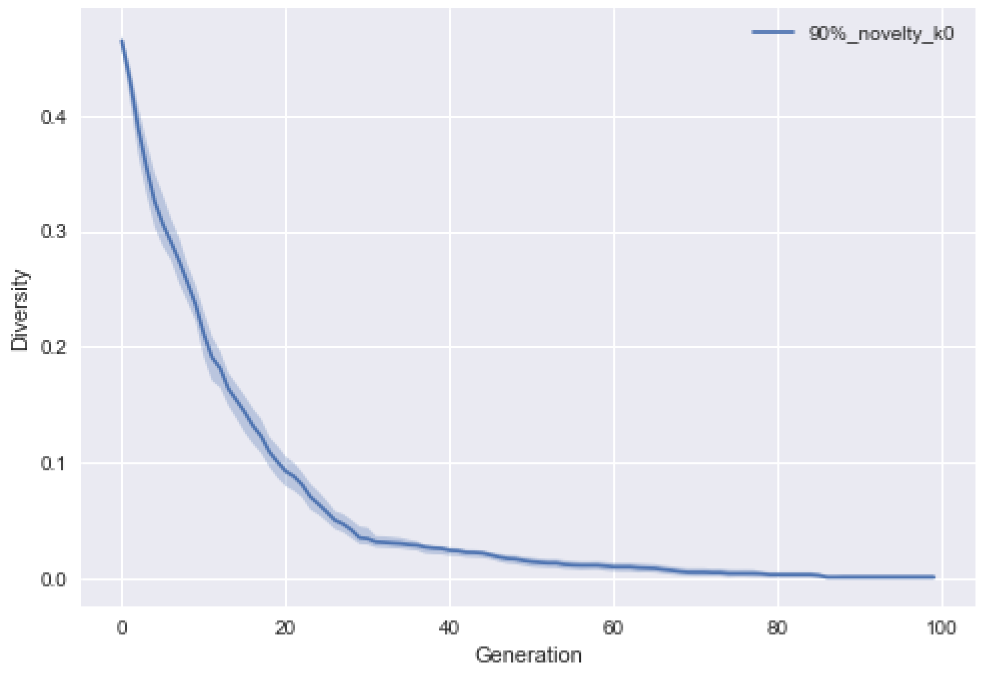
In [122…

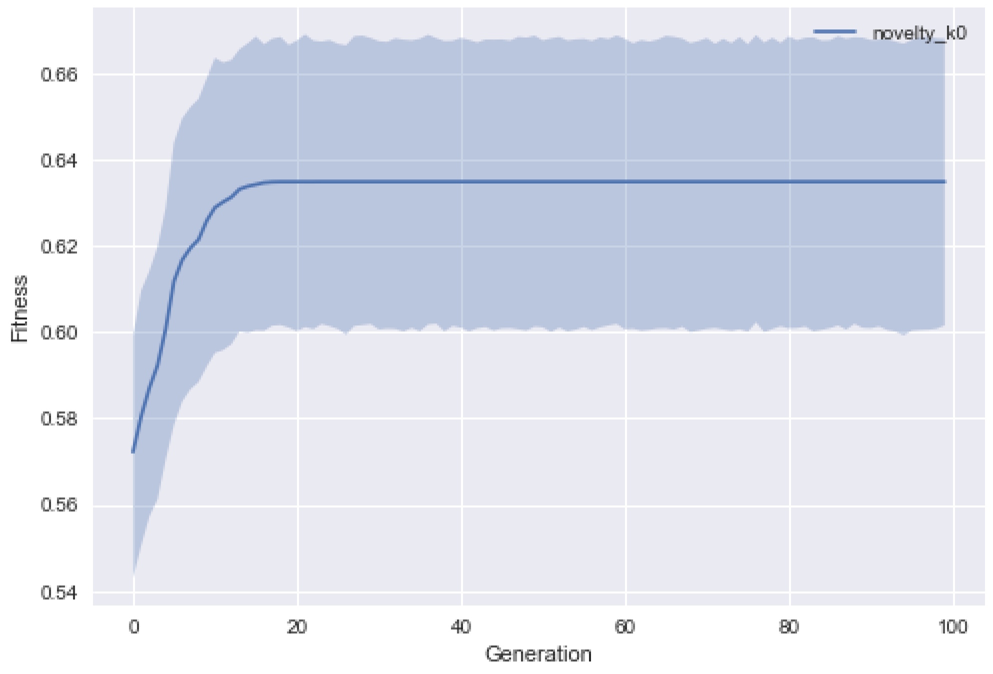
```
# plotting
plot_mean_and_bootstrapped_ci_over_time(input_data = experiment_results, name = ["90%_nove
plot_mean_and_bootstrapped_ci_over_time(input_data = diversity_results, name = ["90%_novel
plot_mean_and_bootstrapped_ci_over_time(input_data = experiment_results, name = ["novelty_
plot_mean_and_bootstrapped_ci_over_time(input_data = diversity_results, name = ["novelty_k
plot_mean_and_bootstrapped_ci_over_time(input_data = experiment_results, name = ["fitness_
plot_mean_and_bootstrapped_ci_over_time(input_data = diversity_results, name = ["fitness_k
```

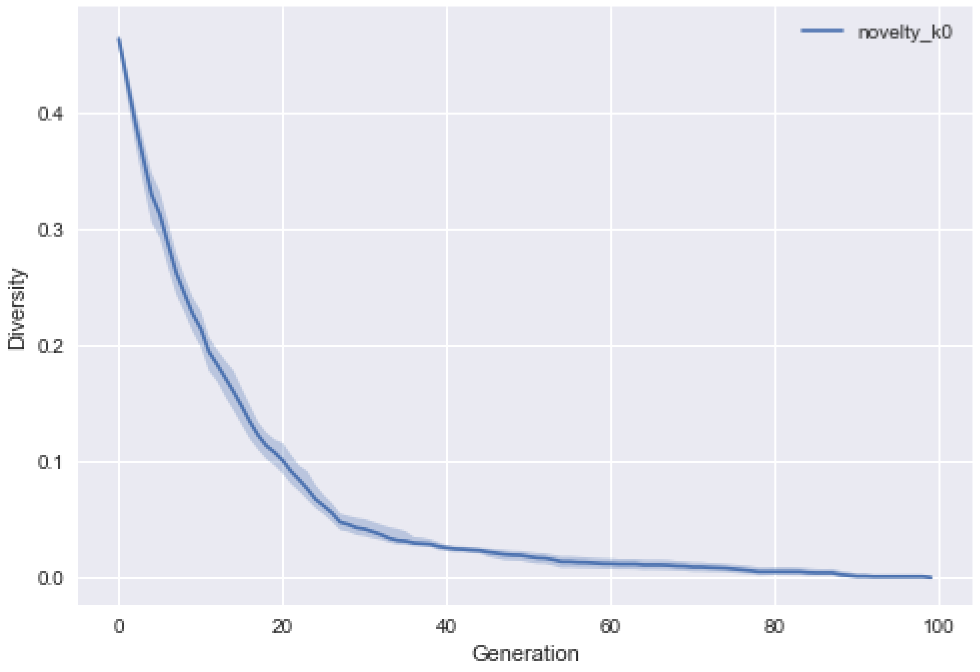
```
plotting 90%_novelty_k0
0
10
20
30
40
50
60
70
80
90
plotting 90%_novelty_k0
0
10
20
30
```

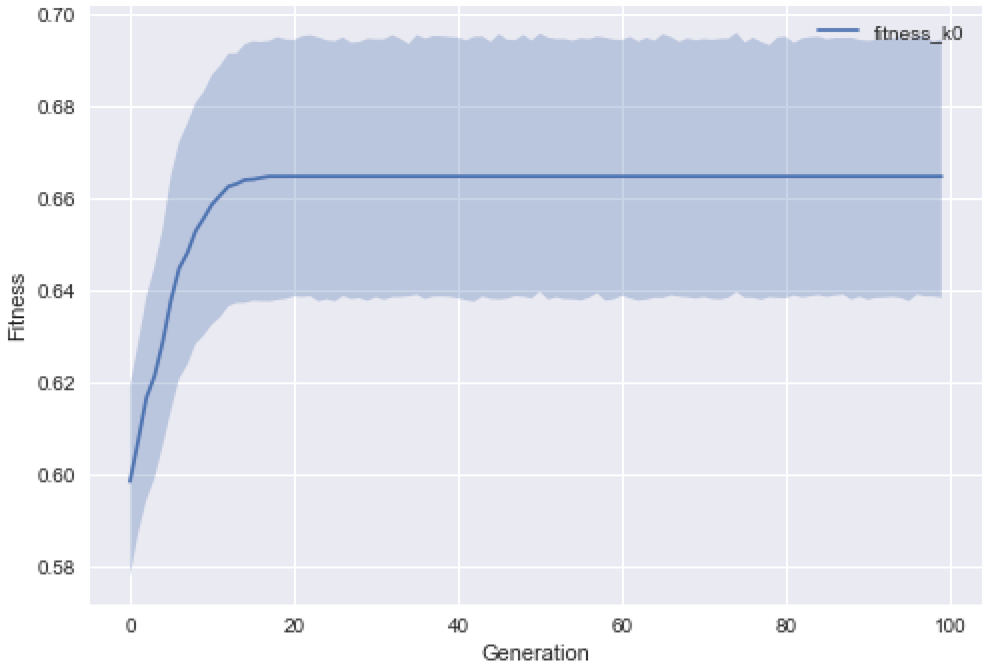
```
40
50
60
70
80
90
plotting novelty_k0
0
10
20
30
40
50
60
70
80
90
plotting novelty_k0
0
10
20
30
40
50
60
70
80
90
plotting fitness_k0
0
10
20
30
40
50
60
70
80
90
plotting fitness_k0
0
10
20
30
40
50
60
70
80
90
```

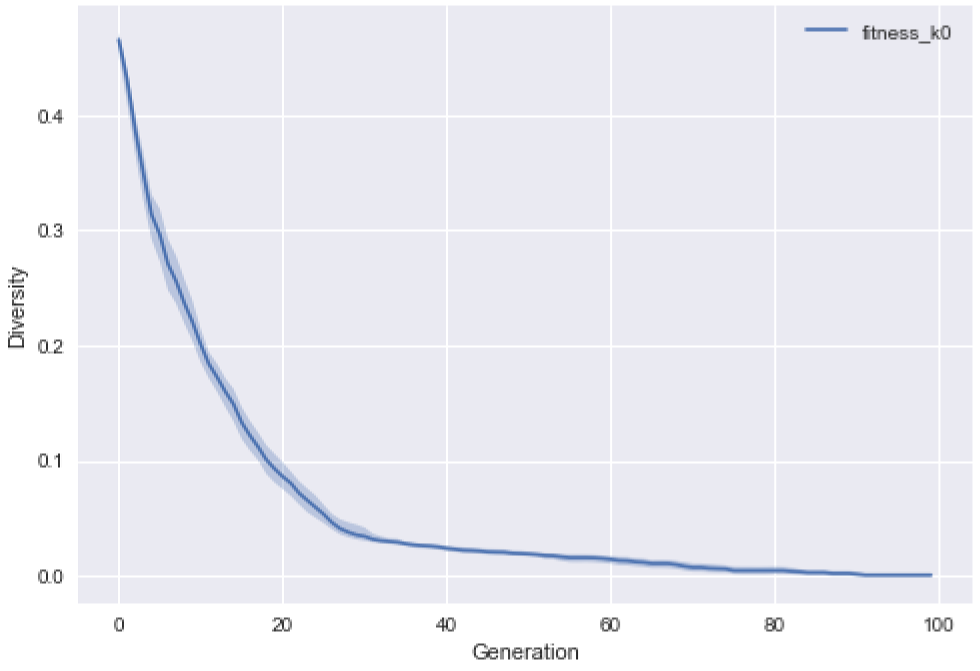

90%_novelty_k0
Fitness
0.56
0.58
0.60
0.62
0.64
0.66
0.68
0
20
40
60
80
100
Generation


90%_novelty_k0
Diversity
0.0
0.1
0.2
0.3
0.4
0
20
40
60
80
100
Generation


novelty_k0
Fitness
0.54
0.56
0.58
0.60
0.62
0.64
0.66
0
20
40
60
80
100
Generation


novelty_k0
Diversity
Generation
0.0
0.1
0.2
0.3
0.4
0
20
40
60
80
100


fitness_k0
Fitness
Generation
0.58
0.60
0.62
0.64
0.66
0.68
0.70
0
20
40
60
80
100


fitness_k0
Diversity
Generation
0.0
0.1
0.2
0.3
0.4
0
20
40
60
80
100

Q12b: Analysis

Were you right? Was novelty, or novelty+fitness helpful? Was it harmful? What was the effect on diversity?

The fitness-based approach does seem to work best here with regards to overall fitness in the 100th generation. All of the charts for fitness and diversity look very similar with diversity almost 0 by the 80th generations and fitness reaching very close to the same values across all of the cases. This seems very odd. It seems that injecting novelty in this situation is slightly harmful.

Q13: Future Work

In this assignment we explored just one (very simple) way to combine novelty and fitness, how else might you want to do this that could be more effetive (and why)?

We could maintain novel subpopulations that compete based on fitness within their own groups. This could help by maintaining several highly fit, yet very diverse subpopulations of solutions.

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:

Just me

In [ ]:
```
```