Jordan White
CS-6613-01
02/24/2021

Project 1

Part A: kNN Algorithm

Abstract

In section A of this project, I implemented a kNN algorithm to create a Supervised Learning program in Python 3.0.

About kNN Supervised Learning

The kNN algorithm is a Supervised Learning algorithm whose goal is to predict what class some item belongs to based on its input data. The algorithm works by comparing the given item's input to the input of many other stored items, then determining the k most similar stored items, and finally predicting that the class of the item in question is the same as that plurality/majority class of the k most similar stored items.

Algorithm Parameters and Implementation: 'labeled-examples.txt' file

I was given 1000 example items, each with two input variables, and one class corresponding to either 0 or 1. The number of groups I would test separately was N = 5, so I divided the 1000 items into five groups of 200 items each. This allowed me to test each of the five groups independently, while storing the other 800 through the 'Store All' or 'Store Errors' methods.

<u>Store All</u>

For the 'Store All' storing method of the kNN algorithm, I had to store all 800 example items and consider each one as a possible nearest neighbor to the test items. I measured similarity between 'neighbors' by calculating the Euclidean distance of the input of each element in the test group against the input of each element of the example group.

In my program, k=5. This means, for each test item, I had to store the five most similar example items. I then determined the majority label of these five nearest neighbors, then added that majority label to a list of predictions. The index of each prediction corresponded to the index of the test item it attempted to predict.

Once I had a list of 200 predictions, I tested each of the other four groups and stored predictions for each one. In the end, I had 1000 label predictions, one for each item in the file. I proceeded to calculate the accuracy of my predictions by comparing them to the established labels of each item; See my accuracy in the 'Results' section.

Store Errors

Afterwards I tested the 'Store Errors' storing method, in which, out of the 800 example items, I only stored the first five items belonging to each class, then I stored whatever remaining items were misclassified by the classifier. I then used whatever items were stored as an example set, and I determined the k closest inputs from this shorter list. This method significantly reduced memory costs, but at the cost of some accuracy (about 5% lower).

Algorithm Parameters and Implementation: 'shuttle.tst' file from ics.uci.edu

To further test the flexibility of my program on more complex data, I downloaded another data set from https://archive.ics.uci.edu/ml/datasets/Statlog+%28Shuttle%29. This data set contains 14,500 items, of which I used 5000. I still broke the data into five groups of 1000, and tested each one at a time against the other 4000. Each item in this data set contains nine input variables, and each item can belong to one out of six possible classes. I tested my kNN algorithm using the 'Store All' storing method on this data. One noticeable fact about this data is that about 80% of the items belong to class '1', so the accuracy rate of my program must be much greater than 80% to ensure it is performing better than chance.

Differences from labeled-examples data set

This data was slightly more complex to work with than the previous data for a couple reasons. First, I had to figure out a way to calculate input data similarity (nearness) based on nine input values instead of two. To accomplish this, I decided to take the square root of the sum of the squared difference of each of the corresponding input values between each pair of items being compared. In simpler terms, I generalized the Euclidean distance formula for nine values. Unfortunately, the ranges of possible values for each of the nine input values differed between each other, but since each input value differed from the others by no more than a factor of 100, I decided against weighting the differences to balance the disproportionate effect inputs with larger ranges.

The second reason this data set was more complex to work with is that I had to modify my code to determine the *plurality* class of the five nearest neighbors, instead of the *majority*, because each item could be one of six classes instead of just two. To accomplish this, I simply created a function to return the most common element in a list and passed the nearest neighbors to that list to find the plurality.

Results

The average accuracy of my kNN Store All algorithm on the labeled-examples data set was roughly 95.1% (Figure 1).

The average accuracy of my kNN Store Errors algorithm on the labeled-examples data set was slightly lower, at around 90.8% (Figure 2). The lower memory cost makes the decision between using Store All and Store Errors a tradeoff between accuracy and memory.

The average accuracy of my kNN Store All algorithm on the shuttle data set was the highest, averaging 99.3% (Figure 3). I think the reason my accuracy was so high is because of two factors: For one, I had more data to work with (4000 stored items as opposed to 800), and secondly, each item contained much more input data (nine values as opposed to two), which likely increased the precision of each prediction.

Conclusions

I now know the basic premise behind Supervised Learning algorithms such as kNN. The methods I learned programming this part of the project could generalize to many other more complex Artificial Intelligence projects, such as image recognition or statistical regression.

If you wish to run my code yourself, simply navigate to the /project1/ folder in your terminal and execute the main.py script. This script uses four other files:

1. NearestNeighbor.py, with most of the primary functions for running this program.
2. file_io.py, to read in the data given to us at the start.
3. Item.py, which contains the Item class I used to abstract each line of data.

Appendix

```
Accuracy of Store All on labeled-examples data for Test Group 1 :  0.925
Accuracy of Store All on labeled-examples data for Test Group 2 :  0.965
Accuracy of Store All on labeled-examples data for Test Group 3 :  0.95
Accuracy of Store All on labeled-examples data for Test Group 4 :  0.97
Accuracy of Store All on labeled-examples data for Test Group 5 :  0.945
Average accuracy of Store All on labeled-examples data : 0.951
```

Figure 1: Accuracy of the predictions of my kNN algorithm using the ‘Store All’ storing method on the labeled-examples data set. Notice accuracies are given for each of the five groups, and the average accuracy is also given.

```
Accuracy of Store Errors on labeled-examples data for Test Group 1 :  0.89
Accuracy of Store Errors on labeled-examples data for Test Group 2 :  0.935
Accuracy of Store Errors on labeled-examples data for Test Group 3 :  0.92
Accuracy of Store Errors on labeled-examples data for Test Group 4 :  0.89
Accuracy of Store Errors on labeled-examples data for Test Group 5 :  0.905
Average accuracy of Store Errors on labeled-examples data : 0.908
```

Figure 2: Accuracy of the predictions of my kNN algorithm using the 'Store Errors' storing method on the labeled-examples data set. Notice accuracies are given for each of the five groups, and the average accuracy is also given.

```
Accuracy of Store All on shuttle data for Test Group 1 :  0.9949899799599199
Accuracy of Store All on shuttle data for Test Group 2 :  0.991991991991992
Accuracy of Store All on shuttle data for Test Group 3 :  0.9939879759519038
Accuracy of Store All on shuttle data for Test Group 4 :  0.9919839679358717
Accuracy of Store All on shuttle data for Test Group 5 :  0.993993993993994
Average accuracy of Store All on shuttle data : 0.9933895819667363
```

Figure 3: Accuracy of the predictions of my kNN algorithm using the 'Store All' storing method on the shuttle data set. Notice accuracies are given for each of the five groups, and the average accuracy is also given.

Part B: Simulated Annealing

Abstract

In the Simulated Annealing portion of Part B, I used Simulated Annealing to search a three-dimensional graph for the absolute minimum point.

About Simulated Annealing

The purpose of Simulated Annealing is to start a search in some state of a problem, and then travel from state to state until the goal state is reached. The algorithm accomplishes this by selecting a new state in the neighborhood of the current one, and always traveling to that state if it is closer to the goal; If the state is worse, SA will travel to it under a certain probability. This probability increases the higher the global 'temperature' is and decreases the worse that new state is compared to the current state. The global temperature slowly decreases over time, and when it reaches zero, the current state is returned as the predicted goal state.

Algorithm Parameters and Implementation

The goal of my algorithm was to locate the absolute minimum in a three dimensional graph. I made a generic simulated annealing algorithm that took a 'schedule' parameter and a 'problem' parameter. The schedule is a list of decreasing numbers all the way to zero, and the problem contains the current state and all other possible states. Each state contains an x-value and a y-value within the bounds of the graph. My algorithm loops through the schedule, constantly searching for the next states in the graph. When a better state is found, it switches to it, and when a worse state is found found, it switches with probability:

$$e^{\frac{(\triangle E)}{t}}$$

where delta-E is the change in fitness of the new state, and t is the temperature. When the schedule reaches its last element (zero), the current state is returned. I tested this algorithm on two functions given to us in ga_eval.py: sphere(x) and bump(x). I also tested two different schedules on each search: One schedule was a linear decrease in temperature, and the other was an exponential decrease.

Results

The minimum value my algorithm found for Bump() using the linear decrease schedule was roughly (0,0) (Figure 4). The minimum value my algorithm found for Bump() using the *exponential* decrease schedule was also roughly (0,0) (Figure 4).

Looking at the graph of Bump() (Figure 5), these minimum values appear to be correct.

The minimum value my algorithm found for Sphere() using the linear decrease schedule was roughly (0,0) (Figure 3). The minimum value my algorithm found for Sphere() using the *exponential* decrease schedule was also roughly (0,0) (Figure 3).

Looking at the graph of Sphere() (Figure 5), these minimum values appear to be correct.

Conclusions

I now know the basic premise behind simulated annealing algorithms and can use it for optimization/search problems.

To run my code for this portion of this project, simply navigate to the /project1/partB/simulated_annealing/ folder in your terminal and execute the simanneal.py script. This script uses two other files I made:

1. schedule.py, which contains the schedule class.
2. problem.py, which reads in the functions and tracks the states.

Appendix

```
Minimum state found for Bump Function using linear-schedule simulated annealing: [0.0, 3.907921733617672e-162]

Minimum state found for Sphere Function using linear-schedule simulated annealing: [2.6477910808957015e-161, 0.0]

Minimum state found for Bump Function using exponential-schedule simulated annealing: [1.621362439704653e-162, 1.1240195390821799e-210]

Minimum state found for Sphere Function using exponential-schedule simulated annealing: [1.2039480165543213e-222, -2.508090367191401e-162]
```

Figure 4: Minimum values found for Sphere() and Bump() using both the linear and exponential schedules (notice the very low E values, approximating each values to 0).

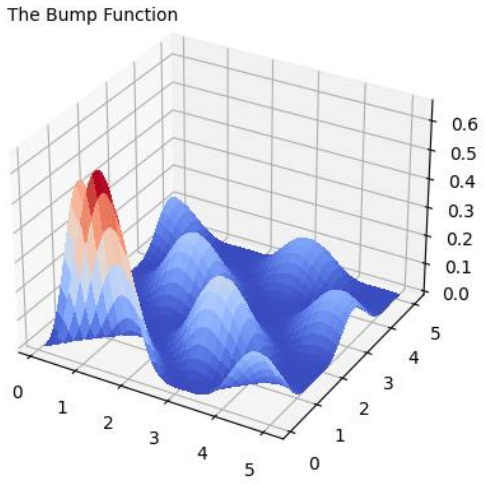

Figure 5: Bump() function

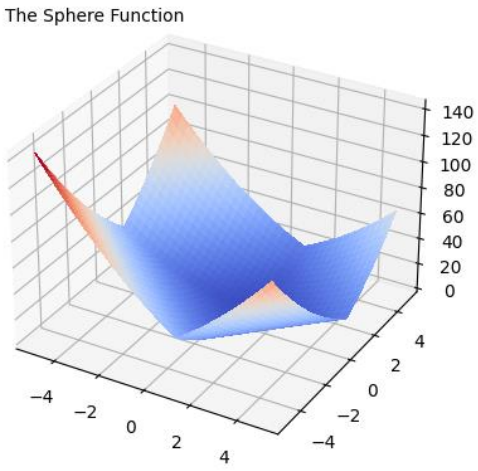

Figure 6: Sphere() function.

Part B: Genetic Algorithms

Abstract

In the Genetic Algorithms section of Part B, I used populations of Bit Strings that evolved over many generations to determine the minimum points on several three dimensional graphs.

About Genetic Algorithms

One purpose of Genetic Algorithms is to evolve new, fitter generations of input from past generations until the new input generates some desired goal state. This can be done using chromosomes, or strings of data, often in bit string form, which represent each individual in a generation. The bit strings that correspond to the fittest output are given a greater probability of 'reproducing' their genetic data into future generation, through greater chance of selection for parenthood. Parenting a child entails an organism splicing its bit string with that of another. The resulting organism also has a small chance of 'mutating' some of its bits, to add genetic diversity.

Algorithm Parameters and Implementation

Main Algorithm

My algorithm was trying to locate the absolute minimum in several three-dimensional graphs. I made a generic genetic algorithm that took several parameters, such as number of parents, mutation chance, and number of generations. This function initialized a population of 500 organisms. Each organism had two bit strings of length 52, corresponding to the x and y input values. These bit strings were then converted into floats for each organism, and the inputs were entered into the function in question. Organisms were stored with their resulting values. A new generation was then constructed one organism at a time. The parents of each new organism were selected from the previous generation. A random sample of ten organisms from the old generation was selected, and the two with the lowest value were selected as parents. Their x-bit strings were spliced in a random spot, as were their y-bit strings, to make the x and y values of the new organism. There was also a small (0.001%) chance that one bit would mutate in the x and y bit strings. This process was repeated for 500 generations, and at the end the most fit organism of the final generation was taken to be the predicted minimum value of the graph.

I ran this algorithm on four of the provided functions: 'sphere', 'shekel', 'langermann', and 'bump'. I created a unit test function, my algorithm is capable of running on all of the functions (it would take very long, however). To accomplish this, one only needs to add the name of the desired function as a string to the ‘functionNames’ list in the main() method.

Alternative Mutation Rate

I also tested my genetic algorithm using three alternate mutation rates on the sphere function. Since I already used 0.1% chance on four functions, I tested out 0.0%, 1.0%, and 100.0% mutation chances on just the sphere function.

Results

Four functions

Predicted minimum value of genetic algorithm on bump(): 0.0 (Figure 7)

This value seems correct when compared to the graph of bump() (Figure 5).

Predicted minimum value of genetic algorithm on langermann(): -1.055 (Figure 8)

This value seems correct compared to the graph of langermann() (Figure 11).

Predicted minimum value of genetic algorithm on shekel(): -4472946 (Figure 8)

This value seems correct compared to the graph of shekel() (Figure 12).

Predicted minimum value of genetic algorithm on sphere(): 0 (Figure 10)

This value seems correct compared to the graph of sphere() (Figure 6).

Modified Mutation Rates

The predicted minimum value of my genetic algorithm on sphere(), using a mutation rate of 0.0%: 0.01 (Figure 13).

The predicted minimum value of my genetic algorithm on sphere(), using a mutation rate of 10.0%: 0.008 (Figure 14).

The predicted minimum value of my genetic algorithm on sphere(), using a mutation rate of 100.0%: 0.0005 (Figure 15).

There seems to be a positive trend between the mutation rates and the fitness of the final state.

Conclusions

I now know the basic premise behind genetic algorithms. This is a very versatile algorithm with many applications. Since the concept of state space searching is so general and so powerful, I am sure this technique will come in handy in the future.

I also found that the greater the mutation rate I use in my genetic algorithm, the closer the end result is to the absolute minimum (the goal state). Although I have not rigorously proven this to hold in all cases, it does make sense to me that such a relationship would exist, as greater mutation seems to open the door to more possible states than those entailed in the first generation.

To run my code for this portion of this project, simply navigate to the /project1/partB/ folder in your terminal and execute the main.py script. This script uses three other files I made:

1. genetic_algorithm.py, which contains most of the primary functions for this algorithm.
2. unit_tests.py, which runs all the tests on the four functions.
3. organism.py, which contains the code for generating/handling bit strings.

Appendix

```
Function: bump
    Number of iterations of Genetic Algorithm: 500
    Fittest bit-string pair: ['0101111101010111011110100011010011000111111
    Resulting input: [ 0.37242854870408615 ,  0.026853356695245356 ]
    Resulting output on bump function: 0.0
```

Figure 7: Predicted minimum of Genetic Algorithm on Bump().

```
Function: langermann
    Number of iterations of Genetic Algorithm: 500
    Fittest bit-string pair: ['1100001100000000000000000000000000000000
    Resulting input: [ 0.76171875 ,  0.5984972422666237 ]
    Resulting output on langermann function: -1.0550880753501979
```

Figure 8: Predicted minimum of Genetic Algorithm on langermann().

```
Function: shekel
    Number of iterations of Genetic Algorithm: 500
    Fittest bit-string pair: ['1110000110010011010111111111011110111111
    Resulting input: [ 0.8811550121172045 ,  0.7236022949218774 ]
    Resulting output on shekel function: -4472946.1310677165
```

Figure 9: Predicted minimum of Genetic Algorithm on shekel().

```
Function: sphere
    Number of iterations of Genetic Algorithm: 500
    Fittest bit-string pair: ['1001100101111111111111101111111101011
    Resulting input: [ 0.5996093152498332 ,  0.599973678588867 ]
    Resulting output on sphere function: 1.7389413856196145e-05
```

Figure 10: Predicted minimum of Genetic Algorithm on sphere().

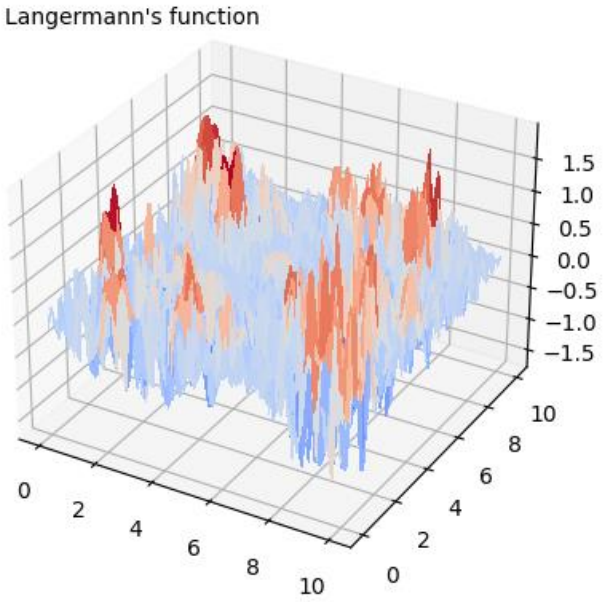

Figure 11: Actual graph of Langermann().

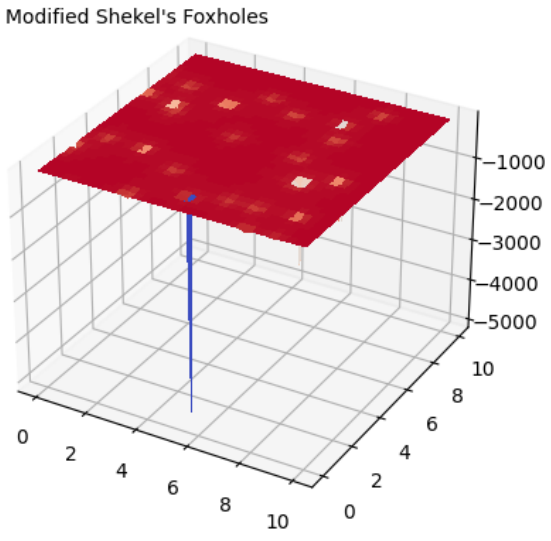

Figure 12: Actual graph of shekel().

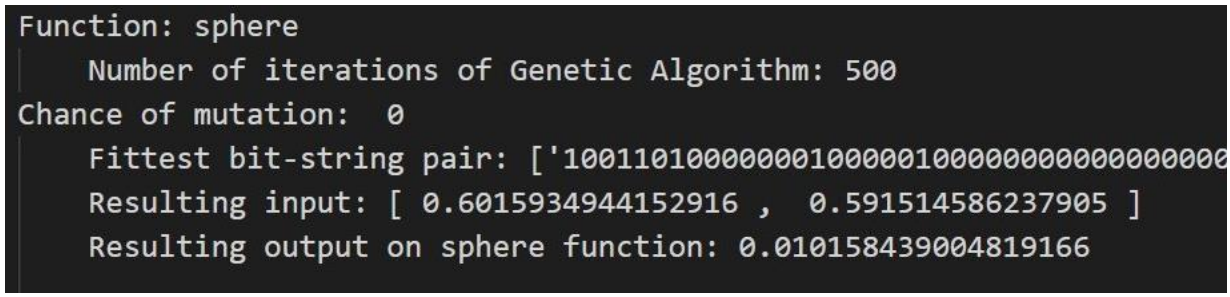

Figure 13: Predicted minimum of genetic algorithm on sphere() using a 0.0% chance of mutation.

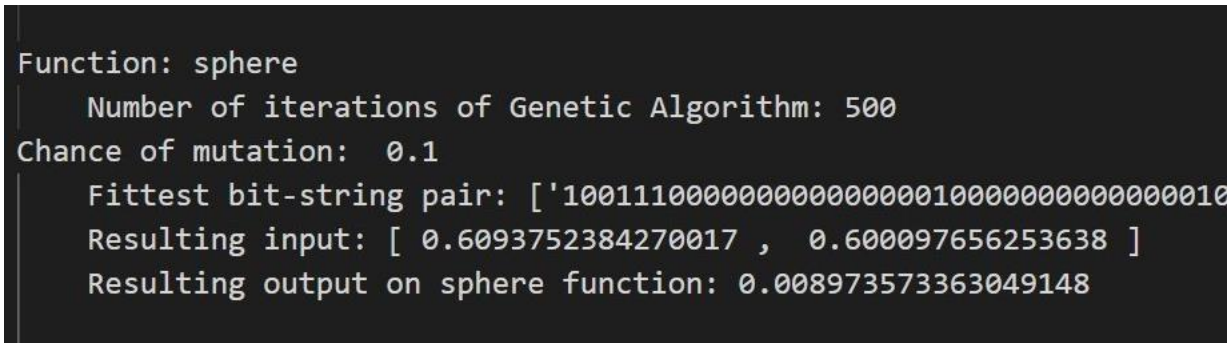

Figure 14: Predicted minimum of genetic algorithm on sphere() using a 10.0% chance of mutation.

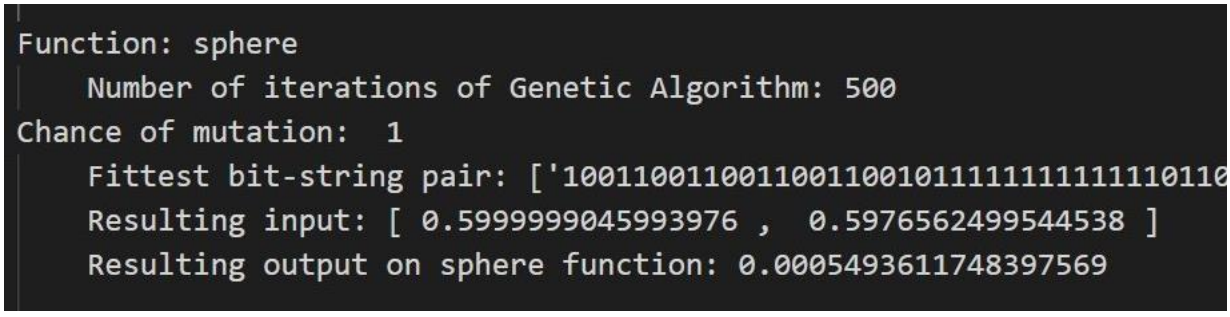

Figure 15: Predicted minimum of genetic algorithm on sphere() using a 100.0% chance of mutation.