Joe Shymanski 2/18/21

Project 1: Adaptation

(Nearest neighbor, Genetic Algorithm & Simulated Annealing)

Objective

The objective of this project is to implement a K-Nearest Neighbors (KNN) algorithm, test it on several different data sets using N-Fold Cross-Validation, implement the local search algorithms of Simulated Annealing (SA) and Genetic Algorithm (GA), and test them on several different functions and common problems in computer science, such as the Travelling Salesman Problem (TSP).

Implementation

K-Nearest Neighbors

KNN accepts three inputs: k, training data, and testing data. The skeleton of the algorithm is subsequently composed of three parts: calculate the distance between the current testing point and every other training point, store the k-nearest training points to the test point, and then calculate the winning label from these k training points. This is repeated for every test point and the predictions are returned.

Before KNN can be executed, the N folds for cross-validation must be generated from the provided training data. To ensure each fold has equal amounts of points, the training data is looped over N times and every Nth point is stored as a test point, the rest are stored as training points. The data is shuffled before this procedure in order to avoid patterns in the storing of data. It is sometimes normalized as well.

In every cross-validation fold, a model is trained based on the current fold's training data. There are two different modes of training a model: storing all the training points and storing only

misclassified training points. The first method will likely yield increased accuracy in its predictions, but it takes up a significant amount of space and time. The second method stores the first k training points of each label and then runs the KNN algorithm on each subsequent training points. If the point is misclassified, it is stored as part of the model. This method will yield decreased accuracy in its predictions, but it makes up for it by taking significantly less time and space than the first method.

Once the model has been built, it is safe to pass into the KNN algorithm as the training data in order to start making predictions on all of the original training and testing data of that fold. This will yield a number of predictions, and an accuracy can be obtained for each subset of data by adding up the correct labels and dividing by the size of the subsets. This can be averaged across the N folds to obtain the average training and testing classification accuracy for the given dataset and training method.

Simulated Annealing

This algorithm, when operating on a simple mathematical function, accepts a function to optimize, a range of values for each dimension a point in the domain can have, a string denoting whether the function should be minimized or maximized, an annealing schedule, an initial temperature, and the number of steps for the algorithm to take. Initially, SA generates a single, random point within the intervals of each dimension of the domain, along with the functional value returned at that point. Then, looping a specified number of times, calculate the current temperature based on the current iteration number, the initial temperature, and the predetermined number of loops. This is done according to the specified annealing schedule, of which I included three: linear, quadratic, and trigonometric.

After calculating temperature, a successor is obtained within a neighborhood of the current point. I did this by generating a Gaussian distribution in each dimension, centered at the value of the current point for that dimension with a standard deviation of one-fifth of the range of that dimension. The algorithm must ensure this successor point is within the bounds of the domain. Then a functional value is calculated at this new point.

Finally, the current and new functional values are compared. If the new point yields a better value, then it replaces the current point. If it is worse, the point is taken with a probability determined by the Boltzmann distribution. After the specified number of steps have been completed, SA returns the current point and current functional value.

For TSP, the algorithm has a slightly different implementation. Instead of a function to optimize and a list of ranges for each dimension, it accepts a list of city locations as an input, as well as all the other aforementioned inputs. The function to optimize is assumed to be the route cost. The initial route is stored as the default permutation of cities and the current cost is calculated from it. When creating a successor route, one of two actions are performed by a coin flip: swap two randomly selected cities with each other or place one randomly selected city directly behind another in the route. The rest of the algorithm is the same, and it returns the current route cost as well as the current route.

Genetic Algorithm

This algorithm takes the same first three inputs as SA, as well as the mutation probability, population size, and number of generations. Initially, GA builds a random population of n-dimensional points using 52-bit strings which can be converted to a float between zero and one, which can in turn be scaled using the specified ranges. Then the algorithm enters its loop, with each iteration representing a generation.

The first action in each generation is to calculate the fitness of each individual based on the functional value returned by the point that the individual represents. These fitness values are used as weights in the process of choosing parents for reproduction. However, the weights must first be manipulated to be greater than zero. Then they are ready to be used for mate selections.

Once weights have been determined, a second population is created by choosing two parents according to their weights, reproducing a child using those two parents, mutating the child with a certain probability, storing the child, and repeating until a second population the same size as the first has been created. Then the second replaces the first and the generation is complete. Once the final generation has been generated, GA will compute the fittest individual from the population. It will then return this individual point as well as its corresponding fitness.

Reproduction simply involves choosing a random index, splicing a segment from the first parent's bit string up to that index, then combining it with the complementary segment from the second parent's bit string. This is done for every dimension of the point. Mutation is even simpler. One randomly selected bit of each dimension is flipped.

For TSP, the algorithm is, much like SA, slightly different. It replaces the function and range inputs with the list of cities, and the rest are the same. The initial population consists of randomly shuffled versions of the given list of cities. Then the fitness and weights are determined by the route cost. The best fitness and route from the final population is returned.

The biggest difference comes in the methods of reproduction and mutation. In reproduction, a crossover point is randomly selected and the child is initialized as a copy of the first parent. Then every city before the crossover point in the child is replaced by the city at the same index in the second parent. In order to ensure the route satisfies the constraints of the

problem, each displaced city from the child is swapped with the city that displaced it within the child route. Mutation simply swaps two randomly selected cities in the route.

Results & Analysis

K-Nearest Neighbors

Table 1

Store All (All) vs. Store Errors (Errors), Normalization (Norm) vs. No Normalization

Storage	Norm	Training Set Accuracy (%)	Testing Set Accuracy (%)	Time Elapsed (s)
		train-file		
All	Yes	98.40	95.80	1.51
All	No	98.20	96.00	1.42
Errors	Yes	94.35	92.50	0.30
Errors	No	94.17	91.50	0.39
		leaf.csv		
All	Yes	90.22	68.24	0.22
All	No	89.12	60.29	0.21
Errors	Yes	90.07	62.94	0.17
Errors	No	88.38	53.24	0.23
		accent-mfcc-data-1.csv		
All	Yes	94.38	80.87	0.21
All	No	94.60	79.65	0.20
Errors	Yes	87.92	72.35	0.08
Errors	No	91.41	76.00	0.12

Note. k = 3 and N = 5 for all runs.

As expected, the Store Errors method performs worse than the Store All method. It is also evident in the train-file data that the Store Errors method dramatically increases the speed of the algorithm on large datasets. For the leaf.csv dataset, there is a significant increase in performance when the data is normalized as opposed to when it is left alone. The rest of the datasets do not show significant changes in accuracy between normalized and unnormalized data. This is likely due to the fact that they are already more or less normalized. Normalization has a negligible effect on time elapsed for KNN. In fact, the most time consuming portion of the algorithm comes

from calculating the distance between each pair of points. Normalization and the Store All method will be enabled in further tests of the KNN algorithm for the sake of accuracy.

Table 2

Varying Values of N

N	Training Set Accuracy (%)	Testing Set Accuracy (%)	Time Elapsed (s)
		train-file	
5	98.28	95.10	1.46
10	98.37	95.90	2.79
15	98.41	95.61	4.15
20	98.39	95.90	5.65
25	98.40	95.70	7.00
		leaf.csv	
5	90.15	66.76	0.17
10	90.95	69.41	0.38
15	90.80	69.12	0.64
20	90.82	70.00	0.83
25	90.77	70.88	0.98
		accent-mfcc-data-1.csv	
5	94.38	77.20	0.15
10	94.63	81.16	0.36
15	94.68	80.53	0.52
20	94.70	82.10	0.73
25	94.71	81.45	0.91

Note. k = 3, Store All = True, and Normalize = True for all runs.

As Table 2 clearly shows, time increases linearly as N increases. The effect N has on KNN performance is less obvious. As N increases, prediction accuracy seems to be largely unaffected, except for the testing sets of leaf.csv and accent-mfcc-data-1.csv. They demonstrate a slight increase in prediction accuracy. However, since the increase is not consistent among datasets and not particularly large, N will be set to 5 for the remainder of KNN runs in order to save on time.

Table 3

Varying Values of k

k	Training Set Accuracy (%)	Testing Set Accuracy (%)	Time Elapsed (s)
		train-file	
3	98.33	95.20	1.26
5	97.97	95.20	1.24
7	97.08	94.10	1.20
9	96.75	94.70	1.42
11	96.58	95.00	1.32
		leaf.csv	
3	90.44	67.35	0.17
5	83.90	64.12	0.22
7	79.71	65.00	0.17
9	77.21	60.88	0.18
11	73.82	63.24	0.19
		accent-mfcc-data-1.csv	
3	94.38	82.66	0.18
5	90.73	79.65	0.16
7	89.06	78.11	0.17
9	86.63	78.43	0.20
11	84.35	78.42	0.19

Note. N = 5, Store All = True, and Normalize = True for all runs.

Varying the value of k does not seem to affect the time taken by the algorithm. However, increasing values of k seem to generally decrease the accuracy of the KNN predictions, most notably in the leaf.csv training data. For this reason, a value of 3 is assigned to k for all other KNN test runs.

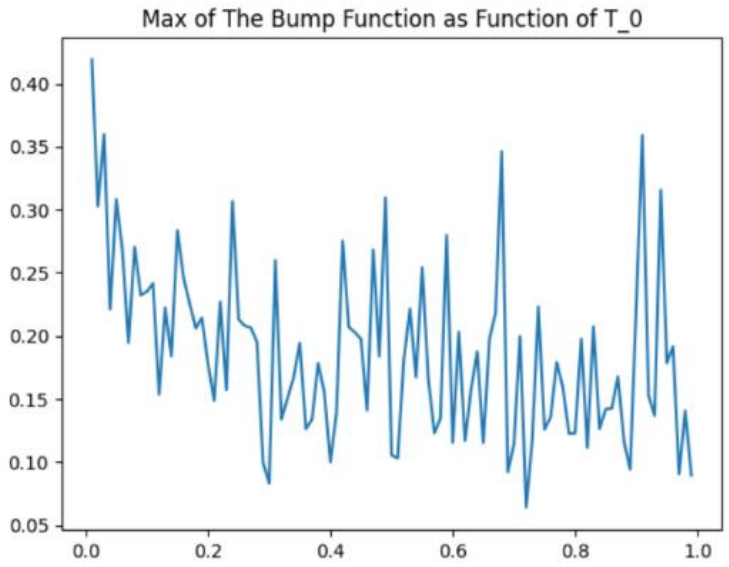
Local Search Algorithms

For SA, increasing the number of steps the algorithm takes will intuitively increase its performance along with the amount of time it takes to obtain a solution. Similarly, increasing both population size and the number of generations in GA will lead to increased performance and time elapsed. For these reasons, the effects that these parameters have on performance will not be investigated further.

For SA, the quadratic annealing schedule performs the best of the three implemented annealing schedules. It slightly edges out the trigonometric schedule on average, and both significantly outperform the linear schedule. Though it is not captured in a subsequent figure, this fact is the reason the quadratic schedule is used for further testing of SA.

Figure 1

Varying T_0 in Simulated Annealing

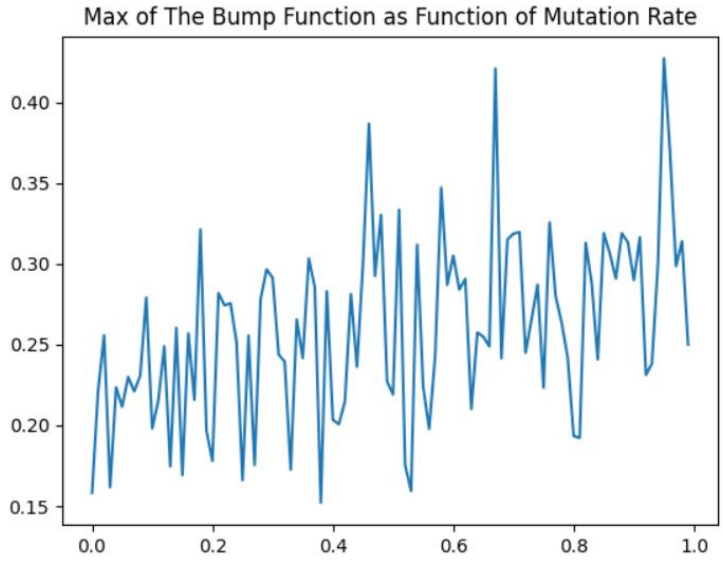


Note. Annealing Schedule = Quadratic and Number of Steps = 100 for all runs. Each T_0 represents an average of five runs.

As you can see from Figure 1, the smaller the initial temperature for SA when operating on the Bump function, the better the average result. This happens to be the case for most of the functions provided. Modified Shekel's Foxholes shows too much variation to discern whether the relationship is positive or negative. Thus, the initial temperature chosen for testing all seven of the provided functions is 0.1.

Figure 2

Varying Mutation Rate in Genetic Algorithm



Note. Population Size = 15 and Number of Generations = 20. Each Mutation Rate represents an average of ten runs.

Figure 2 demonstrates that increasing mutation rate will roughly increase the performance of GA on the Bump function. This general trend holds for all seven provided functions. For this reason, further tests of GA on functions will use a mutation rate of 1. This is likely due to the fact that a mutation only changes a single bit of each dimension's bit string, which likely does not throw the point completely off and leads to more detailed exploration of nearby areas. This is likely the primary reason that GA outperforms SA on Modified Shekel's Foxholes, which has many local minima with tiny areas in the domain.

Table 4

Simulated Annealing (SA) vs. Genetic Algorithm (GA) on Functions

Function	Optimum	Average Value		Average Time Elapsed (s)	
		SA	GA	SA	GA
Sphere	Min	0.0045	0.0776	0.29	0.33
Griewank	Min	0.0347	0.1286	0.43	0.40
Shekel	Min	-1.1×10^{4}	-6.7×10^{13}	2.73	2.22
Michalewitz	Min	-1.8636	-1.8518	0.47	0.44
Michalewitz	Max	1.8379	1.7802	0.45	0.43
Langermann	Min	-1.5886	-1.4855	5.46	4.25
Langermann	Max	1.7616	1.5981	5.33	4.21
Odd Square	Min	-1.0218	-0.9527	0.32	0.33
Odd Square	Max	1.1398	0.9771	0.32	0.38
Bump	Max	0.6527	0.6340	0.49	0.48

Note. For SA, Annealing Schedule = Quadratic, T_0 = .1, and Number of Steps = 10000 for all runs. For GA, Mutation Rate = 1, Population Size = 85, Number of Generations = 90 for all runs.

Table 4 demonstrates that, with optimal inputs that execute each algorithm in about the same amount of time, SA on average outperforms GA. The only exception occurs with Modified Shekel's Foxholes, and I believe the reason for this is the precision in GA's mutations. Since each bit string is 52 bits long, and mutation only changes a single bit at a time, the algorithm can introduce incredibly small changes that have significant consequences in this specific function.

Table 5

Simulated Annealing (SA) vs. Genetic Algorithm (GA) on Travelling Salesman Problem

Number of Cities	Average Route Cost		Average Time Elapsed (s)	
	SA	GA	SA	GA
5	2.2310	2.2310	0.40	0.39
10	2.9056	2.9593	0.42	0.38
15	3.3473	3.7004	0.42	0.42
20	4.1633	5.2467	0.47	0.47
30	4.4343	7.4173	0.67	0.58
40	5.2981	10.9187	0.78	0.65
50	6.2117	13.7021	0.87	0.78
75	8.0728	21.9947	1.13	1.05
100	9.9866	31.7943	1.38	1.38

Note. For SA, Annealing Schedule = Quadratic, T_0 = .5, and Number of Steps = 100000 for all runs. For GA, Mutation Rate = .01, Population Size = 180, Number of Generations = 200 for all runs.

For TSP, the current implementations have drastically different performances as the number of cities increases. SA performs quite well as more and more cities are introduced. However, GA seems to fall apart as more than 20 or 30 cities make up the route. Graphically, the solution routes generated by SA are generally convex with very few segment crossovers. The GA solution routes have significant amounts of path intersections and are clearly suboptimal. The reason for this is unclear to me. Perhaps the reproduction implementation for TSP hurts more often than it helps to approach a solution.

Conclusion

KNN performs a bit better when using the Store All method as opposed to the Store Errors method but takes significantly longer on large datasets. Normalization can be used to increase performance unilaterally on datasets which may have large range discrepancies between variables. The value of N should generally be set low – around 5 or 10 – since the performance is roughly the same and the time taken is minimal. The value of k should be odd and generally low, in order to avoid ties and yield the best accuracy.

For local search algorithms, the current implementation of SA generally outperforms GA on both functions and TSP. The difference is much smaller on functions than it is on TSP. SA performs optimally using the quadratic annealing schedule, a small value for initial temperature, and a large value for number of steps. GA performs best on functions when mutation rate, population size, and number of generations are all high. These last two parameters also work best when they are roughly equal. However, a lower mutation rate is best for TSP.