LZSCC.361 – Coursework Part II: AI Project Report

38555123

Dec 15, 2023

1 Task 1 - Classifier

1.1 Implementation

The K- nearest Neighbour Classifier(KNN) takes a k value, training data set, training data label, and a point as parameters and performs the following steps:

- Calculate the Euclidean distances of all points in the training data set and the given point, storing the values and associated indices.
- Sort the distances based on their values and select the top k values (nearest neighbors).
- Evaluate the most frequent label among the neighbors using their indices on the training data label.
- If k is even and there is a tie, arbitrarily predict label 1.
- Return the predicted label.

Euclidean distance was chosen over Manhattan distance due to lower errors observed with their respective best possible k values.(see section 1.2)

1.2 Evaluation

The classifier's evaluation is based on a function measuring the error ratio by computing miss-classified data points over all data points. For different k values, the classification errors on training and validation sets are computed:

- $k = 3$:
  - classification error on training set: 0.01667
  - classification error on validation set: 0.03519
- $k = 7$:
  - classification error on training set: 0.02778
  - classification error on validation set: 0.0463
- $k = 19$:
  - classification error on training set: 0.0025
  - classification error on validation set: 0.05185

- $k = 31$:
  - classification error on training set: 0.03333
  - classification error on validation set: 0.05185

k=3 is included because it is the best value of for k. This was observed by going over possible values of k for tset and evaluating the error and finding k with the smallest error value

Additionally, these are the errors when using Manhattan distance are provided.(k=4 is the best possible error)

- $k = 4$:
  - Training set error: 0.01667
  - Validation set error: 0.03519
- $k = 7$:
  - Training set error: 0.02778
  - Validation set error: 0.0463
- $k = 19$:
  - classification error on training set: 0.0025
  - classification error on validation set: 0.05185
- $k = 31$:
  - classification error on training set: 0.03333
  - classification error on validation set: 0.05185

2 Task 2 – Regression with Genetic Algorithm

2.1 Implementation

Encoding: The final Genetic Algorithm (GA) is a continuous GA using real valued type genes representing the coefficients [a0, a1, a2] for the chromosomes. I decided on this approach based on the information in [1] suggesting that it is best to use a continuous GA for problems that involve working with continuous variables. The main reasons being that: continuous GAs require less storage and are inherently faster of the absence of a decoding step. Moreover, I considered the fact that it would be more convenient for me to both write and evaluate the functions used in the GA. I knew that issues related to limited floating point precision were a risk but was prepared to change the encoding if necessary but the GA works fine as is.

Objective Function: The mean square error is used as it is a common error measurement that is documented to work well. Additionally, it helps to have a normalised value to understand how far off points are from a functions surface.

Fitness evaluation: To calculate the fitness of an chromosome, an array of predictions is first created using the (x, y) pairs of the training set. This array in passed to the objective function along with the an array of the actual z values of the training set

Selection Strategy: Roulette wheel selection is used so as to bias selection away from individuals with bad fitness while still maintaining a level of diversity. The fitness values of the

population are inverted as (1/fitness) first since the objective function is an error measure that ought to be minimized and the low fitness values are better.

Crossover Strategy: An intermediate arithmetical approach is employed to explore and exploit the possible search space between the boundaries of each gene as much as possible. The crossover probability is set to 0.75 because it was experimentally determined to be effective (more on this in section 3.3).

Mutation Strategy: An adaptive mutation strategy is used in the final GA because it was experimentally observed to work well. The mutation probability is set to 0.02 and increased if the best fitness does not change over successive generations.

Population Initialization: The GA creates 10 chromosomes since it is was experimentally shown to be a good value that works without increasing run-time of the GA. The chromosomes are made by using the random.uniform() to choose numbers for a0,a1 and a2 within their respective set boundaries. When doing so I add 0.0001 to the upper ranges so as to accurately represent the actual boundaries of the coefficients since the random.uniform() function does not include the upper bound and it was not allowed to import sys.

New population creation: The new population going on to a successive generation is made up of the children of the selected parents and 3 elite members within a population. This was also determined experimentally. The size of any population is always going to be half of the original population so as to avoid overpopulation over generations.

Termination Strategy: The GA stops and returns the best chromosome of a population of the final generations if the best fitness of the generation is below 2.98, an experimentally determined value that has been observed to guarantee termination while also to producing significantly low classifications errors on the given data sets.

2.2 Experiments

Arriving at the GA described above involved developing different versions of GAs that built upon each other. Seven versions were developed and for each experiments were carried out. The GA's behaviour was observed by using a function experimentOnGAname() to take in the parameters of the GA and run the GA 5 times to control for the effect of randomness. The average fitness of those runs, the plot representing the change in the best fitness withing a generation, and the best fitness and best chromosome of every generation were printed out.

Firstly though, to see how each parameter affected the GA several experiments were done where all parameters were kept constant and one variable. This is how the values for parameters were chosen and what was meant by "experimentally determined" in the previous sections. To see the effect of population size for example

```
def runExperimentOnVanilla():

    experimentOnVanilla(10,0.02,0.8,10)
    experimentOnVanilla(15,0.02,0.8,10)
    experimentOnVanilla(25,0.02,0.8,10)
    experimentOnVanilla(35,0.02,0.8,10)
    experimentOnVanilla(45,0.02,0.8,10)
    experimentOnVanilla(45,0.02,0.8,10)
```

```
[[3.973575676722151, [5, 0.02, 0.8, 10]],
 [4.75166593477435, [15, 0.02, 0.8, 10]],
 [4.585174504663837, [25, 0.02, 0.8, 10]],
 [4.9862219792481985, [35, 0.02, 0.8, 10]],
 [4.9180809638456795, [45, 0.02, 0.8, 10]],
 [5.272451620233104, [55, 0.02, 0.8, 10]]]
```

Figure 1: Example experiment on population parameter

2.2.1 Version 1: "Vanilla" GA

This was is most rudimentary version of the GA. Parameters: population size, mutation probability, crossover probability and number of generations as parameters. Given these values, it initializes a population, selects parent pairs using roulette wheel selection, uses An intermediate arithmetical approach for crossover, uses uniform mutation to create new children that makeup the new population for the next generation. As a termination strategy, it runs for the given number of generations but stops early if the best fitness of a population is less than 1. Insights gained after experimentation:

- Increasing the population does leads to worse in fitness at high values and also increases the run-time of the GA.
- High values for crossover probability lead to better fitness
- The fitness gets worse with longer generations
- GA does not behave consistently between runs
- GA never terminates before set generation number
- GA the lowest fitness it can get is 4.287
- GA the chromosomes with the lowest finesses are likely to have at least 1 if not all of the genes set to zero, suggesting that that they are getting stuck at a false stop Changing the parameters of the GA did not significantly change the inconsistent behaviour of the GA. Here is 3 out of 5 runs some plots of the GA

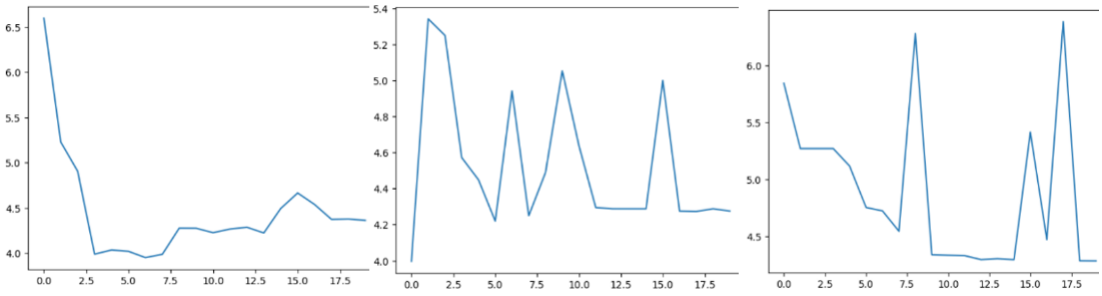

Figure 2: 3 out of 5 runs of the version 1

2.2.2 Version 2

Change from previous version: the termination strategy is that the GA runs until the difference of the best fitness between successive generations is below a certain threshold. Insights gained after experimentation:

- Increasing the stagnation limit increased the run-time of the GA and does not have a significant boost in fitness it after values greater than 6
- Best fitness is around 4.25 but still get 0 gene values
- It is better than version I in terms of consistency but still room for improvement

2.2.3 Version 3

Change from Previous version: mutation probability that sets the mutation at 0.02 and increases it if the average fitness is greater than 3. Insights gained after experimentation :

- Fixed the issue of the 0 gene values but is not guaranteed to terminate all the time for stagnation limit more than 1
- Worse values for fitness

2.2.4 Version 4

Change from previous version: the termination strategy is to stop when diversity check within a population is below a certain threshold Insights gained after experimentation:

- Better average fitness
- Bad consistency and a lot of changes in fitness over the generation

2.2.5 Version 5

Change from previous version: Add elitism

- Better average fitness but bad chromosomes
- For the GA to terminate the threshold would have to be high, defeating its purpose

2.2.6 Version 6

Change from previous version: Revert back to the termination strategy of version 3 and 2. Insights gained after experimentation:

- A significantly more constantly behaving GA
- Best Fitness is around 2.95
- The minimal effective number for the stagnation limit is 3

2.2.7 Version 7 final

Change from previous version: termination strategy is to stop when the best fitness is below 2.98 and change mutation rate based on the stagnation of the fitness over generations. Insights

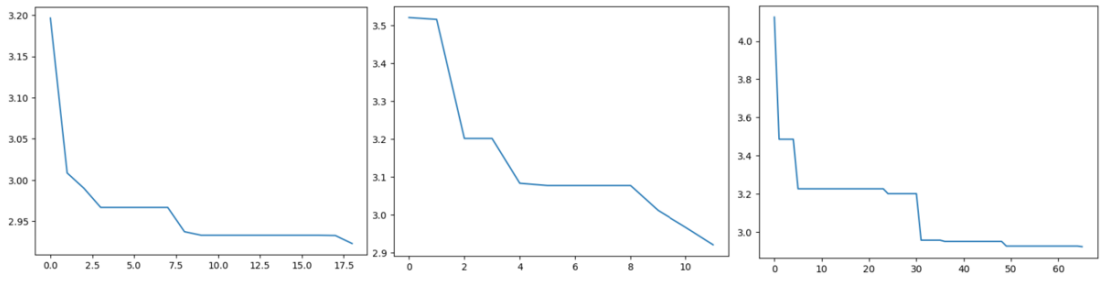

Figure 3: 3 our of 5 runs for final GA

gained after experimentation:

- The fitness decreases over generations
- It is guaranteed to terminate but high variability on which number of final generation, range seems to be from as low of 5 to 220

3 Comparative Analysis

The classification error for the best values for k(k=3) are 0.01667 and 0.035 for the training. Using that as benchmark, for comparison the GA classifier and can do better than these values for both data sets but it is not always guaranteed. It never significantly get worse compared to the KNN as well The following are examples values of the classification error based for different outputs of the GA. The time it takes to run the GA has a lot of variability, the worst recorded worst case being 54 seconds.

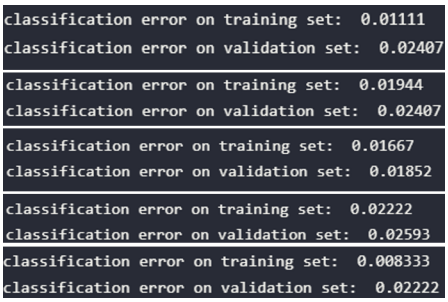

Figure 4: Classification error based on different values of GA

4 Conclusions

The GA classifier displays potential for better classification errors than KNN but lacks consistent performance and exhibits varied runt-times especially in worst-case. Further optimization of GA parameters might enhance reliability and efficiency in producing improved classification results while reducing run-time variability. In my opinion, if a use case scenario requires a fast solution with a sufficient enough accuracy, KNN is best suited over GA. However, if one seeks to explore the maximum amount of the solution space with little concern for algorithm hyper tuning and run time, then one should opt for a GA.

References

[1] Randy L. Haupt. *Practical genetic Algorithms.* John Wiley Sons, Inc., Hoboken, New Jersey, 2nd ed. edition, 2004.