Artificial Bee Colony (ABC) Algorithm for Regression

COMP6000 Project

Bradley Fuller

bf84@kent.ac.uk

School of Computing

University of Kent

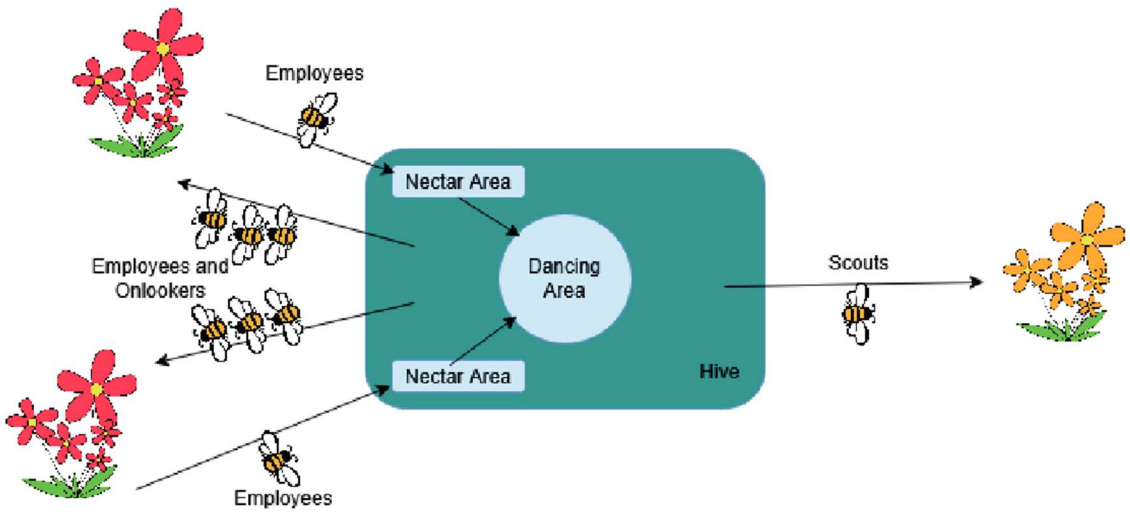
22nd March 2025

1 Abstract

This report will go into detail about the concept, development and uses of the Artificial Bee Colony algorithm, especially, as the title suggests, its use in solving regression problems. The algorithm itself is inspired by the foraging behaviour of honeybees, such as their exploration and exploitation strategies. Due to the nature of what it is based on, it has the benefit of being understandable by those with knowledge in this field, and those who do not. It will also prove its use and effectiveness in function optimisation, with direct examples of its use. The results showing clearly its efficiency in solving the function optimisation problem and the regression problem (a real world application as shown in this report), with the speed of which the correct or optimal solution is found. In this project I have tried the linear regression model but I am aware of other regression models that could be explored in the future, such as the exponential model.

2 Introduction

As stated earlier, the artificial bee colony algorithm is an optimisation algorithm that is based upon honeybee foraging behaviour. The 'bees' are assigned three different roles which in the case of the program, align to different tasks being performed, with food sources being different possible results, as illustrated below. To give a brief explanation for each role, the employee bee takes a known source and searches nearby, meaning to slightly alter its current source and see if it results in a higher fitness, if yes that one is saved, if not it is discarded and it sticks with its original source. The goal of this phase is to broaden the search and hopefully find better results. The onlooker bee phase is more focused, with each bee picking a source at random, with the chance being weighted towards ones of a higher fitness, before just performing the role of employee. The goal of this phase is similar to employee, however with the chance leaning more towards those of higher weighting, it makes sure the better quality sources get explored more, improving them. Finally is the scout phase, which unlike the first two which run for every iteration, only runs after a set requirement is met, that being when a source is 'exhausted'. In the code, when a source is checked for better nearby ones, and nothing better is found, a token on that source slowly goes up, and once it hits the limit, the source is exhausted, and is likely close to, if not the best, in that area. As such, the scout generates and entirely new source in its place, and the employees then start the search again. Obviously the best overall source would be saved on every iteration, which by the end would be returned as the result. The frequency at which these tasks are performed combined with the amount of times it is iterated upon allows it to find optimal solutions very quickly, as will be shown more in the Function Optimisation and Regression sections. On top of this, we also simulate it three times and show the average of the results, due to the level of randomness that comes with the algorithm. To elaborate on the randomness aspect, when the program starts, the food sources are generated randomly within certain boundaries. As such, sometimes a source could be generated that is already very close to the solution, meaning technically it didn't take many iterations, or it could start far from it and take a little bit before finding the optimal solution.



3.1 Function Optimisation

To start with, the objective of function optimisation is to find the values in a function which provide the best result, or the highest 'fitness'. In the case of the functions we tested, the goal in each case was to minimise the objective function, meaning that we wanted to take the inverse of the value, and get the smallest result possible, as the smaller the value the closer it would be to the solution to the function, as such after calculating the fitness for each function, it would be inverted and compared as if it was smaller instead. Now, obviously this projects focus is on regression, however focusing on function optimisation first was beneficial to us when starting off. The main reason being that when using a function to test the algorithm, unlike the regression data we used, we already knew what the result should be. As such, we could tell when the programming was running well as once it was returning results that lined up with the actual solution for the function, we were able to tell the program was working correctly. We started with a simple function, as once the basis is written, changing the code to work for other functions only required editing the function calculation aspect, the algorithm itself would be unchanged.

Once the starting function was being returned correctly, while this in of itself was a good sign, we decided to test further by using multiple 'optimisation test problems'. We took multiple problems from here and implemented them into the function, even keeping all of them and swapping between with a simple switch case, to allow for showing the results of multiple functions later. From here, moving on to regression wasn't too huge of a step, as the ABC algorithm itself was working, though the algorithm function did need very minor changes, the bigger focus was the fitness function, which would need the most changes to work with the regression model we were working with, this will be delved into more in the Regression section. With the results found from the function optimisation we could at least tell how efficient the program could be, with there not needing to be a huge amount of iterations to get close to if not the perfect result, even when taking into account the levels of randomness that comes with the algorithm. In terms of the specific functions chosen, the ones we ended up using were the Beale, Booth, Matyas and Three Hump Camel. These ones were each chosen for their own reasons, the first two were chosen as they were still somewhat simple and not too far from the starting function we used for testing, which was $f(x,y) = (x + y)^2 + (x - y)^2$. The second two being a bit tough to implement and typically taking slightly longer to figure out the optimal solution, albeit not by much however.

3.2 Function Optimisation | Results

Below is the results from running the Beale Function. Obviously to show every single interation would take up too much space, but the results at least show how close the program would get to the optimal solution with not many iterations.

```
Simulation 1...
Simulation 1 Done! Performed 100 iterations.
Simulation 2...
Simulation 2 Done! Performed 100 iterations.
Simulation 3...
Simulation 3 Done! Performed 100 iterations.
Results of Beale Function:
Simulations         Best Solution (x,y) found          Computed f(x,y) of best solution
1                   (2.9054, 0.4799)                   0.0020
2                   (2.8904, 0.4668)                   0.0026
3                   (3.0176, 0.5053)                   0.0001
                    Average Performance                0.0016

[Done] exited with code=0 in 1.049 seconds
```

In the case of the Beale Function, the correct solution is (3, 0.5), so the program is getting close, which is where the computed of best solution part helps to tell. This is figured out by diving one by the best fitness, and subtracting that by one. This changes the result from a fitness score to instead represent roughly how far off the value is from the best solution, as well as taking the average from each simulation. Below is the same result table for the 'Three Hump Camel' function. The answer to which is (0,0)

```
Simulation 1...
Simulation 1 Done! Performed 100 iterations.
Simulation 2...
Simulation 2 Done! Performed 100 iterations.
Simulation 3...
Simulation 3 Done! Performed 100 iterations.
Results of Three Hump Camel Function:
Simulations         Best Solution (x,y) found          Computed f(x,y) of best solution
1                   (0.0000, -0.0000)                  0.0000
2                   (0.0000, 0.0000)                   0.0000
3                   (0.0000, -0.0000)                  0.0000
                    Average Performance                0.0000

[Done] exited with code=0 in 1.393 seconds
```

As stated earlier, while not even a second difference, the three hump camel function does take very slightly longer to iterate over the other function.

4.1 Regression | Data Set

Once the algorithm was functioning correctly for function optimisation, the next part was taking that and using it to find the optimal solution for a regression problem. The data set I used was the Real Estate valuation data from https://archive.ics.uci.edu/dataset/477/real+estate+valuation+data+set. This data set has six features (transaction date, house age, distance to nearest mrt, no. of convenience stores, latitude, longitude) which is used to predict the house price (target variable). Notice that the range of different features varies dramatically, for example the distance to the nearest MRT and the number of convenience stores. Therefore, after importing the data file into the program, I normalise the data set. By normalising, I scale all of the data to ensure its range is between 0 and 1. This was done by going through every column of data, and finding the biggest and smallest in each one. These values are set as the 'min' and 'max' for each, and then normalise the data in each column according to the formula below:

'normalisedData = (data – min) / (max – min)'

The data set has 414 examples, in order to get more accurate prediction results, I put 2/3 of the examples into the training set, and the remaining 1/3 into testing set. The training data is the data that will be input into the ABC algorithm to find optimal weights in the linear regression model, to predict the house price. The testing data is used to evaluate the predictive accuracy of the linear regression model on the house price. While talking about the data set, I thought I would mention the simulation aspect. When the program runs, the ABC Algorithm is performed 3 times fully, and averages are taken, to help account for the randomness factor, however in the case for the data set being normalised and randomly split, this is performed once and used 3 times for the data. I felt it unnecessary to also randomise the data assigning as there is already such a large randomness factor at play with the algorithm itself. This is something I could look into with cross validation which will be touched on more in the Conclusion.

4.2 Regression | Applying ABC

With the data normalised and split, I can now use it within the algorithm itself. As was stated in the function optimisation section earlier, the ABC Algorithm uses 'food sources' to search around and identify potential solutions, in the case of regression, these food sources are the weights for the linear regression model, the formula for which is shown below.

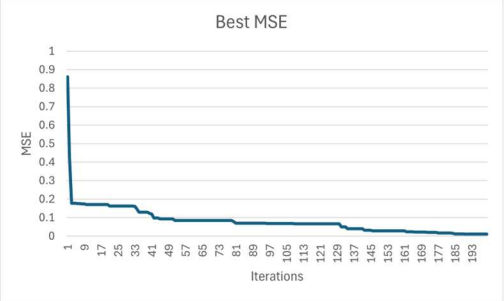
$$y(x1,x2,x3,x4,x5,x6) = w0 + w1x1 + w2x2 + w3x3 + w4x4 + w5x5 + w6x6$$

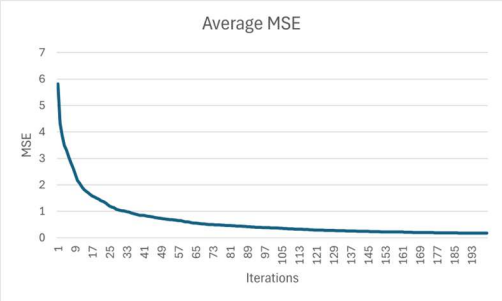
In this case, the x's in the function are the values from the training data set for each column, and as stated the w's (or weights) are the food source from ABC. The vast majority of the ABC algorithm itself plays out much the same as how it does with function optimisation, however I did have to make a few changes for it to work correctly on this function. Changes such as changing food sources to have 6 values instead of 2, increasing the amount of iterations to allow for more accurate results, and making sure the data is read from specifically the training set. The part I had to redo completely, was the fitness function, since I was now working with a completely different function entirely. This function especially, required working with something completely different, while for every function optimisation problem I could have the numbers scored based on their size, in that case since they were all minimisation, it gave better scores to lower numbers, the regression problem is different.

I want to measure how close a predicted value was to the actual value, so while fitness could work here, it was better to work with Mean Squared Error (MSE for short). MSE is useful for finding the difference between two values, especially in that the values getting squared makes it more punishing for larger errors. In the case of this function, the actual value and predicted values are the ones I want to calculate the difference between. This is done by filling in the function with the values being called in the function, so filling in the x's with the features from the data set, the w's being the food sources (predicted values) and the y being the actual value (the cost). Then once we have the two values, just taking the predicted value away from the actual value and squaring the result, which results in the MSE. Now when this is used in ABC, it still requires the number to be larger as higher value is what the 'bees' search for, as such we invert this number with a simple function, (1/(1+mse), which first makes sure the number is between 0 and 1, and helps the 'bees' see whether this is an optimal source. Once all the iterations are done, this same function is performed with the test data and the average MSE is compared to see if the results match up well.

4.3 Regression | Results

The results of the regression function lined up with what I was hoping to achieve with this which I was very happy to see. It is however a little bit more nuanced then just showing the result table. As such, below I have plotted two lines charts, the first one titled; “Best MSE” is somewhat self-explanatory. I took what was considered the best MSE in each iteration from one simulation and plotted it. In this case it was for 200 iterations, which is higher than function optimisation to allow for a more detailed answer, but not as high as it could go as after a certain point it feels pointless as the line does not move much. The second, and more interesting chart is the Average MSE chart, this one, like the best MSE chart, takes from the same simulation and same number of iterations, but instead takes the average MSE of every food source at that point in the iteration. At first glance they will seem quite similar, however looking at the Y axis on both, the average MSE starts from a much higher value, in this case close to 7, whereas Best starts from around 0.9, by the end however, both are close to 0. I feel this well demonstrates both the randomness factor mentioned earlier, with how high the starting point for the average MSE is as many of the early sources will be extremely inefficient, but also shows how efficient ABC can be at finding the optimal solutions, as even towards the end the average is somewhat close to the best MSE, thus, the optimal solution.





To go alongside these charts, below is the result table of the same function. The two charts specifically being based on the data from ‘Simulation 1’.

```
Results:
Simulations         Best Source Found                                        Training MSE          Test MSE
1                   -0.0354, 0.0166, -0.1817, 0.1933, 0.4225, 0.1346, 0.1706        0.0107          0.0157
2                   0.5403, -0.0564, -0.3000, -0.5651, -0.1275, -0.2616, 0.2505     0.0170          0.0244
3                   -0.2097, 0.0914, -0.0391, 0.0653, 0.1172, 0.0670, 0.5653        0.0085          0.0139
                                                                  Averages:         0.0121          0.0180

[Done] exited with code=0 in 2.625 seconds
```

From this result table, we can confirm that even with only 200 iterations, the algorithm is finding solutions that very accurately predict what the price of the house could be. While the best sources found are nice to see, it in of itself doesn’t tell us much, but the Training and Test MSE give insight into how close that prediction is. For example with Simulation 1 again, the training MSE prediction is 0.0107 off of the actual value, and that is after being squared to enhance any errors. This clearly shows an accurate prediction, which for the training data is maybe a bit expected, which is why I have the testing data. The Test MSE is only 0.0157 off, again after being squared. While a bit further off than the Training MSE, its still within decimals of being correct, and this is using sources found via the training data, which proves its use in predicting.

As an extra to include, while I feel the above examples illustrate the efficiency well, below I have included two extra images. Both of these are after upping the number of iterations from 200 to 1000 per simulation, obviously letting the algorithm search for much longer. The first is just a selection of iterations from simulation 1 in this case which I wanted to include to show the jump for average MSE a bit more accurately. As for the second, it is the results of these much longer iterations. While it isn't the perfect judge of efficiency I wanted to include the time the code took to run, as an added way of showing its speed.

```
Iteration Number: 1 | Best MSE: 1.2177142896477235 | Average MSE: 6.455341328614682
Iteration Number: 2 | Best MSE: 1.124431343543768 | Average MSE: 5.438445963241946
Iteration Number: 3 | Best MSE: 0.9236754093681538 | Average MSE: 4.870251801089357
Iteration Number: 4 | Best MSE: 0.9146326453395164 | Average MSE: 4.399231028811036
Iteration Number: 5 | Best MSE: 0.7703737316766992 | Average MSE: 3.848917696993566
Iteration Number: 6 | Best MSE: 0.7703737316766992 | Average MSE: 3.4038982277111067
Iteration Number: 7 | Best MSE: 0.7703737316766992 | Average MSE: 3.024032244926464
Iteration Number: 8 | Best MSE: 0.7703737316766992 | Average MSE: 2.7793311083394676
Iteration Number: 9 | Best MSE: 0.7703737316766992 | Average MSE: 2.642999474528046
Iteration Number: 10 | Best MSE: 0.7627847472559033 | Average MSE: 2.539291576369453
Iteration Number: 11 | Best MSE: 0.5429328755680323 | Average MSE: 2.362575176114073
Iteration Number: 12 | Best MSE: 0.4720620465473009 | Average MSE: 2.2477990432139237
Iteration Number: 13 | Best MSE: 0.4720620465473009 | Average MSE: 2.153019292744582
Iteration Number: 14 | Best MSE: 0.4645723013929035 | Average MSE: 2.038393124542688
Iteration Number: 15 | Best MSE: 0.4293475439443539 | Average MSE: 1.9603967518099985
Iteration Number: 16 | Best MSE: 0.4283564586164019 | Average MSE: 1.897767860751884
Iteration Number: 17 | Best MSE: 0.4283564586164019 | Average MSE: 1.8109894452985182
Iteration Number: 18 | Best MSE: 0.4283564586164019 | Average MSE: 1.7483861903165177
Iteration Number: 19 | Best MSE: 0.26885056924049744 | Average MSE: 1.6851481194005422
Iteration Number: 20 | Best MSE: 0.26885056924049744 | Average MSE: 1.632490824709182
Iteration Number: 21 | Best MSE: 0.26885056924049744 | Average MSE: 1.5601567571219621
Iteration Number: 22 | Best MSE: 0.26885056924049744 | Average MSE: 1.5314832654868176
Iteration Number: 23 | Best MSE: 0.26736095227442025 | Average MSE: 1.4821733775024581
Iteration Number: 24 | Best MSE: 0.26736095227442025 | Average MSE: 1.4656007238493514
Iteration Number: 25 | Best MSE: 0.26026473780212545 | Average MSE: 1.4404886145381122
```

```
Results:
Simulations         Best Source Found                                          Training MSE        Test MSE
1                   0.1659, 0.0515, -0.1072, -0.2512, 0.0855, 0.3707, -0.0681  0.0075              0.0065
2                   0.3018, 0.0211, -0.1206, -0.3546, 0.0558, 0.1501, -0.0244  0.0070              0.0059
3                   0.5543, 0.0360, -0.1044, -0.5672, 0.1228, -0.0949, -0.2601  0.0083              0.0070
                                                                  Averages:     0.0076              0.0065

[Done] exited with code=0 in 5.213 seconds
```

5 Conclusion

In conclusion, I feel I have succeeded in proving the effectiveness in using the ABC algorithm to solve both function optimisation and regression problems. I believe the speed at which the algorithm converges onto the optimal solutions in both are strong evidence of its efficiency. As well as this, the regression model having such low MSE values, especially on the runs with more iterations, shows that its predictions are extremely accurate, even when compared to data it is not trained on at all and even taking into account the inherent level of randomness that comes with using it. I believe its efficiency is one of the major strengths of this algorithm. I would also like to touch on its versatility, it's something I mentioned earlier on in this report, as when the program was first written, its main focus was on the function optimisation aspect, however when I moved onto using it for regression, while the fitness function did change pretty much entirely, the algorithm itself had very minor changes. I feel this is a huge advantage of using this, as I think it wouldn't take too many changes to make it work for other functions.

While I feel the algorithm is efficient and all things considered not very expensive in terms of runtime, I do have to factor that while it adding more iterations did improve results, it does come at the cost of runtime. With the data and functions I was working with this wouldn't be an issue, however I can see that if working with larger data sets or just adding more iterations and simulations, this could start to add up, not even factoring working on more complex models such as the exponential model. While with this data I do not believe it made much of, if not any, difference, the randomness is something that for larger data sets could be more of an issue, and in that situation would need to be looked into. If I were to continue working on this in the future, trying larger data sets and different models and seeing if ABC would work as effectively with those as it did for these ones. As mentioned in an earlier section, I would also like to see if cross validation would have any impact on the result. In this case that would mean instead of using the same sorted data for all 3 simulations, instead, sorting them differently every time, maybe even changing the distribution of examples such as 50/50, or trying with less training and more testing. I feel having those as different simulations then comparing those results would give a good insight to its impact. Less related to ABC, if I came back to this in the future, I would be interested to try different algorithms, such as Particle Swarm and Ant Colony, as I wasn't able to look into those as much as I would've liked for comparison due to time constraints.

Overall, I do feel that the ABC algorithm was efficient in this use case, and did a good job of solving the function optimisation and regression problems, especially with how close the predictions in regression were.

6 Acknowledgements | References

I would like to thank Jie Shao, my project supervisor, for her assistance and patience during this project, with the advice given being a huge factor in my understanding of this concept early on, as well as the Natural Computation lectures from herself and Alex Freitas, the lecture slides I frequently used as a refresher. As well I would like to thank the references attached below, some of which supplied data and others used for insight and research.

Surjanovic, S. and Bingham, D. (2013) *Virtual library of simulation experiments:*, *Optimization Test Functions and Datasets*. Available at: https://www.sfu.ca/~ssurjano/optimization.html (Accessed: 18 February 2025).
Where I obtained the optimisation test functions.

Yeh, I.-C. (2018) *Real estate valuation*, *UCI Machine Learning Repository*. Available at: https://archive.ics.uci.edu/dataset/477/real+estate+valuation+data+set (Accessed: 16 February 2025).
Where I obtained the data set used for regression.