Assignment 2

This document will contain all answers to the theoretical questions in assignment 2 of the *Data Science for Security and Forensics* course. Please check out the .py files as well as the images within the delivery folder for answers to the practical questions.

For task 1 we are working with the following mathematical function:

$$f(x, y) = (x + 2y - 7)^2 + (2x + y - 5)^2$$

1.1: Looking at the function we have defined and plotted. I would suggest that a minima would lie between 1 and 3 (for values of x). If we only consider the provided values of x and y [-10,10] one could argue that this minima is a global minima. However, if we consider all possible solutions for x and y I would think that there are other minima's which go «lower» than our provided function and interval of values.

As for a global optima we don't have a lot to go on. Looking at *Figure 1* we see that there seems to be a "hilltop" at x = -10 however, whether this is an optima is in my opinion impossible to determine. This is however, only if we consider the full set of possible solutions and values for the function. If we instead only consider the provided interval [-10,10] one can argue that there is a global optima at x = -10 and a local optima at x = 10.

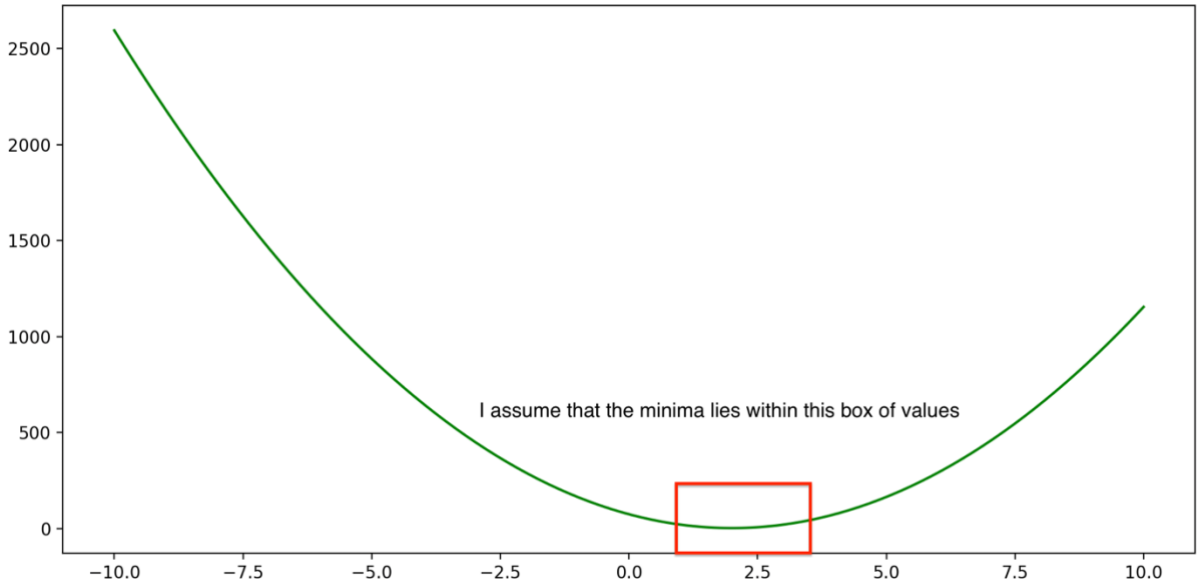


Figure 1 The hypothesized global minima given our interval of x and y values

1.2 After finishing and running my genetic algorithm I get the following results:
I ran the code five times for each configuration and the results are as follows:

Config	Population size	Crossover prob	Mutation prob	Number of generations
Config 1	4	25%	5%	100
Config 2	10	50%	25%	1000

Results Config 1

Coordinates 1	**1,3**	**1,3**	**1,3**	**1,3**
Value 1	0	0	0	0
Coordinates 2	**5,2**	**1,3**	**9,3**	**1,3**
Value 2	53	0	320	0
Coordinates 3	**1,11**	**1,-3**	**1,3**	**1,3**
Value 3	320	180	0	0
Coordinates 4	**1,11**	**1,3**	**1,3**	**3,3**
Value 4	320	0	0	20
Coordinates 5	1,1	4,3	0,3	0,7
Value 5	20	45		

Results Config 2

Coordinates 1	**2,15**	**-2,11**	**2,3**	**8,3**	**5,-9**	**-6,3**	**-3,1**	**-2,-3**	**10,-2**	**2,-7**
Value 1	821	173	5	245	416	245	164	369	170	425
Coordinates 2	**2,-2**	**2,3**	**2,6**	**10,1**	**2,15**	**0,3**	**-10,15**	**-6,2**	**3,3**	**10,7**
Value 2	90	5	74	281	821	5	3809	306	20	773
Coordinates 3	**5,0**	**7,-3**	**6,13**	**4,-6**	**8,1**	**2,-3**	**-6,0**	**8,-3**	**-5,11**	**-10,-10**
Value 3	29	72	1025	234	153	137	458	89	1161	2594
Coordinates 4	**-4,5**	**2,-4**	**3,13**	**2,13**	**-6,3**	**6,-1**	**4,4**	**2,-5**	**-8,2**	**0,-5**
Value 4	65	194	680	585	245	45	74	261	482	389
Coordinates 5	**-1,8**	**8,-1**	**7,-3**	**7,-1**	**10,0**	**2,5**	**13,-12**	**6,-4**	**-6,5**	**-10,-15**
Value 5	65	101	72	68	234	41	405	90	153	3809

1.3 Gradient descent: For gradient descent I decided to test with all the provided parameters. The code has two stopping criteria (see code for full documentation) it stops if rate of change in either point values or function values is too small (in code this is set to 0.001). The following were my testing configurations:

Config	Learning rate	Number of iterations	Stop rate of change point	Stop rate of change value
1.1	1	100	Yes	Yes
1.2	1	100	No	Yes
1.3	1	10 000	No	No
2.1	0.01	100	Yes	No
2.2	0.01	100	No	Yes
2.3	0.01	10 000	No	No
3.1	0.001	100	Yes	No
3.2	0.001	100	No	Yes
3.3	0.001	10 000	No	No

Results

Config	Starting coordinates	Coordinates	Value	Iterations
1.1	(9, 10)	x = 8.32e+123 y = 8.32e+123	4.31e+246	100
1.2	(-9, 3)	x -5.55e+123 y -5.55e+123	1.92e+246	100
1.3	RESULT TOO LARGE	RESULT TOO LARGE	RESULT TOO LARGE	125
2.1	(-3, 5)	x = 1 y = 3	0.000412	33
2.2	(-5, -9)	x = 1 y = 3	0.02337	18
2.3	Terminal deleted the start coordinates	x = 0.9…7 y = 3.0…4	6.3108e-30	10 000
3.1	(-9,3)	x = 1 y = 3	5.654e-06	42
3.2	(5,-6)	x = 1 y = 3	0.0018	27
3.3	Terminal deleted the start coordinates	x = 1.0…4 y = 3.0…4	1.578e-30	10 000

1.4 Looking at the results from the two algorithms we see that with the right configuration both algorithms find a minimum (within our defined field of values likely global).

Genetic Algorithm: For my genetic algorithm **config 1** (populationsize 4, crossover probability 25%, mutation probability 5%,number of generations 100) does find the minima somewhat consistently (see table of results from GA). As I didn't set a stopping criterion for this algorithm it ran all the generations every time however, the convergence rate varied greatly based on the initial candidates. Since I chose roulette wheel selection for my

algorithm, in the instances where the random values were at the local minima or close to it the algorithm converged in one or two generations and the final result different only due to mutation. For this configuration I think 20 generations may be enough given solid start values. I do however think that if I had done a weighted candidate selection process and the pairing process the algorithm would have converged faster (it selects parents purely based on their probability from their roulette wheel value and pairing is random).

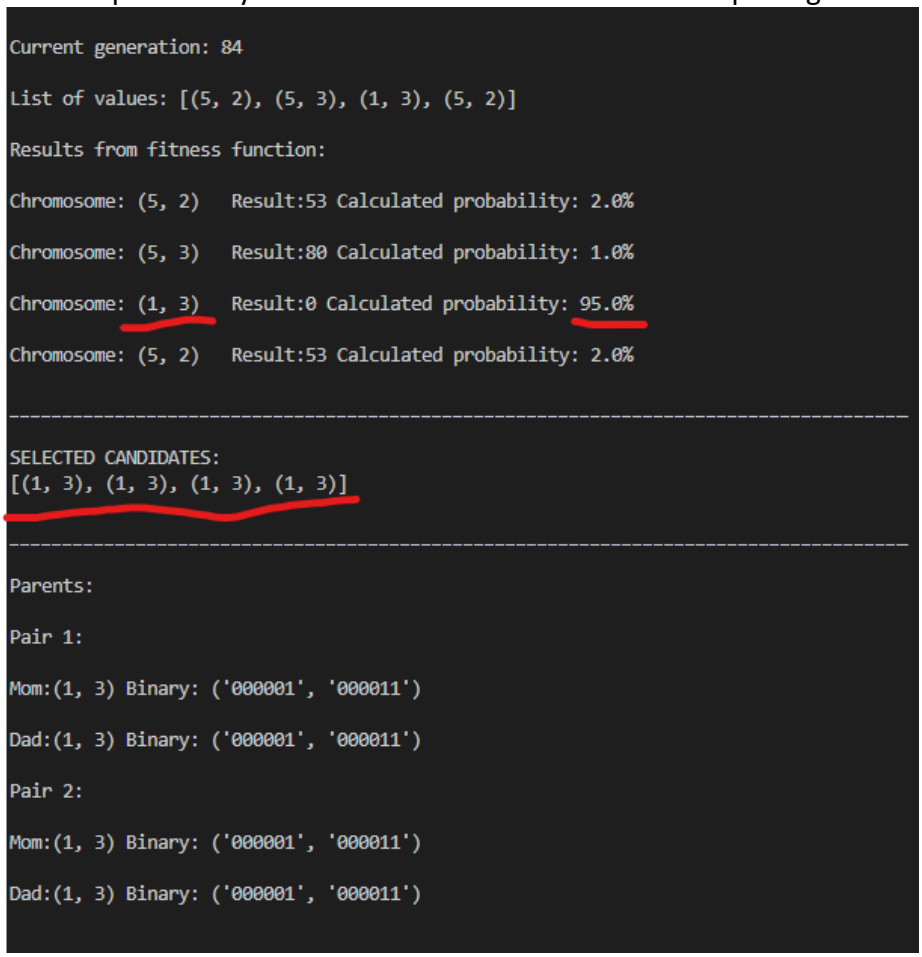


Figure 2 Example of why the algorithm may converge fast, 1,3 gets selected and other probabilities are really low.

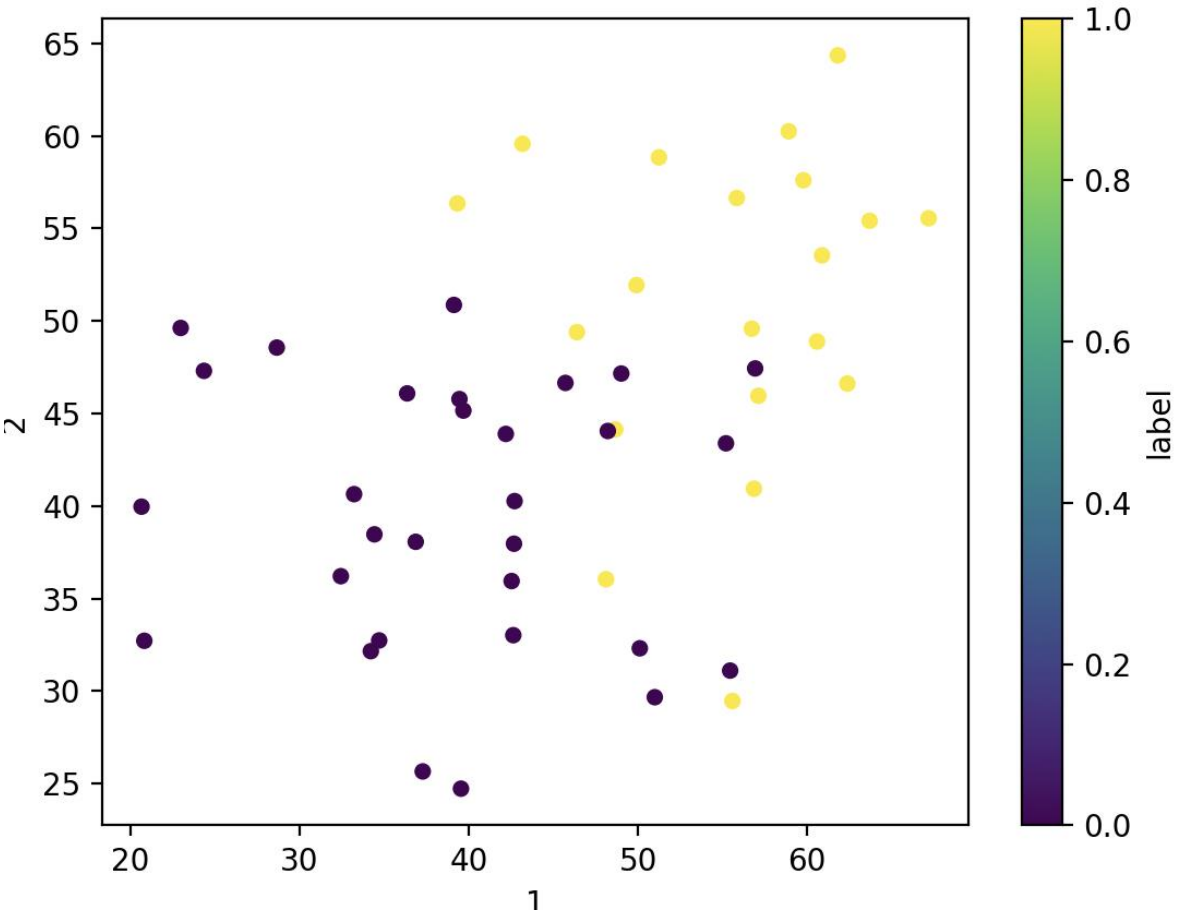
Having experimented with a bigger population size and a higher crossover and especially mutation probability I would advise against these parameters for my algorithm. A higher mutation chance and a high population size only worsened my results (see table of results from **config 2**). This configuration found the minima three times and had on average lesser values close to the minima. I think this is mainly due to the higher rates of change in the genes, the configuration does explore more of the feature space, but it also lacks the ability to converge on low values.

Gradient descent: For gradient descent the best results were produced with a learning rate of 0.01, these tests always found the minima regardless of the number of iterations. I think this configuration performed the best because of its learning rate, it lies in the middle of the other configurations and we see that for learning rates 1 and 0.001 the steps either pass above the minima and diverge to insanely high values or the algorithm takes too small steps

and spend too much time finding the minima. This is further backed by my results showing that **config 2.1,2.2 and 2.3** performed the best with regards to iterations. For my specific codebase the stopping criteria which revolved around rate of change in function values performed the best with a stop at 18 iterations.

2.1 K nearest neighbours clustering: K nearest neighbours (KNN) is an algorithm often used in machine learning. It is based around the proximity of values/pixels or other measurements. Its main purpose is to classify or group data points based on similarities in results and measurements. For classification the algorithm labels datapoints together based on majority vote. It takes data points which are close/similar based on a feature or value and groups them together under a label. In order to group data points we first calculate the distance between points around the datapoint we are looking at for this distance you can use Euclidian, Manhattan, Minkowski etc (Types taken from this article by IBM: https://www.ibm.com/topics/knn). We take the distances of each point and order them, before we take the K closest points and group them together. The chosen distance is the way we measure the distance between points, the method will vary based on the use case.

2.2 For my script I started of by trying to plot all the parameters in the *train.csv* document in a scatter plot to get a broader understanding of the dataset we were provided with. This did not work well as scatter plots need two values two display datapoints. For simple visualization I used column 1 and 2 and got the following plot:



Having plotted the values I needed I started looking around the web for ways to implement the KNN classifier. I ended up using a variety of resources for this, mainly sources stated in

the codebase. I ended up with the scikit learn KNN algorithm. For testing I used the following configurations and ended up with the following results:

Config	K neighbours	Distance metric	Accuracy	Normalized accuracy
1.1	1	Euclidian	60%	80%
1.2	7	Euclidian	80%	90%
1.3	15	Euclidian	70%	90%
1.4	31	Euclidian	70%	70%
1.5	40	Euclidian	50%	50%
2.1	1	Manhattan	80%	90%
2.2	7	Manhattan	90%	90%
2.3	15	Manhattan	90%	90%
2.4	31	Manhattan	60%	70%
2.5	40	Manhattan	50%	50%
3.1	1	Chebyshev	50%	70%
3.2	7	Chebyshev	60%	60%
3.3	15	Chebyshev	60%	90%
3.4	31	Chebyshev	70%	100%
3.5	40	Chebyshev	50%	50%

We can see from our results that for values of K close to half the size of our dataset perform well for most distance metrics. Manhattan performed the best for our dataset with an accuracy of 90% for K values 7 and 15. Very high K values are bound to perform poorly because they take too much data into consideration, if we look at the scatter plot above we see that a small portion of the data points are bundled close together even though they are of different classes, these contribute to misclassification when values of K are too high. The same is true for too low K values, here we sample too little of our solution space which lead to misclassification.

2.3 After normalizing data we see that the accuracy jumps up by around 10 to 20% for all configurations. With one configuration reaching 100%, we can contribute this to the fact that we deal with smaller variations in data. When the model has values within the same range it has an easier time comparing them yielding better results. In our case column seven goes from having values in the 400-500 range which is around ten times larger than the other columns. By rescaling the data this issue disappears!