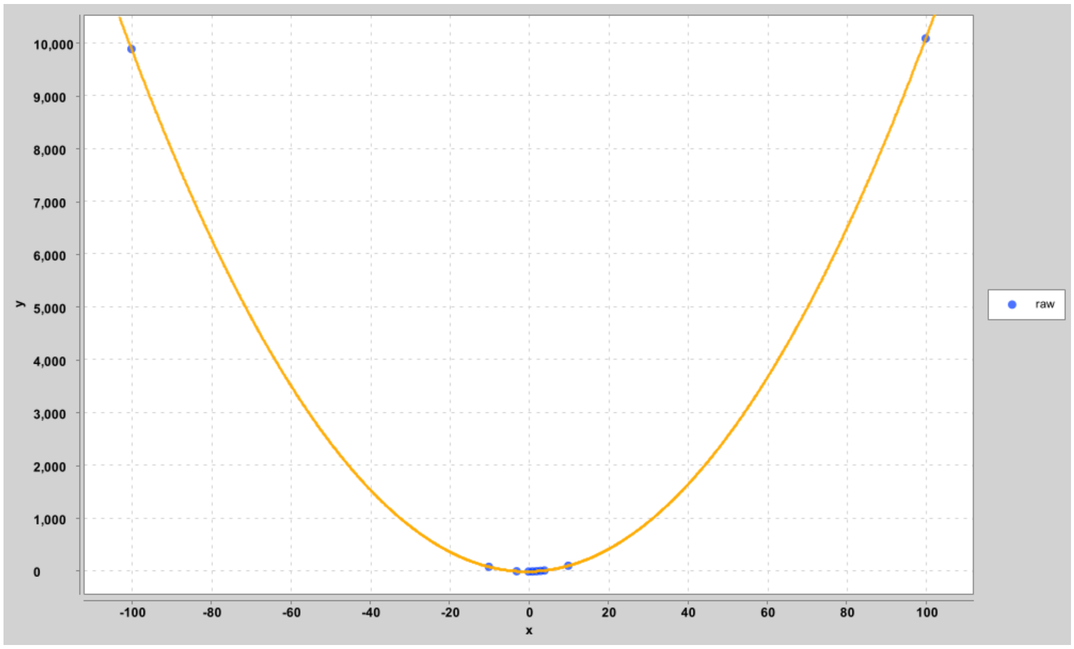
Question One:

Univariate non-linear regression in Java with inital values of w_0, w_1 and w_2 set to 0 and α set to 0.00000001 over 10000 epochs. My code[1] yields the following hypothesis function and the associated weights:

```
>> h(x) = (1.0002793258507536 * x^2) + (0.9820597257019426 * x) + 0.0011254551807739726
>> w2 = 1.0002793258507536
>> w1 = 0.9820597257019426
>> w0 = 0.0011254551807739726
```

After plotting this function (using xchart) we can see parameterized hypothesis function fits the training data very well.



My code also outputs how much the cost changes by each time it is incremented; I have shown the last few lines below:

```
>> ...
>> 0.8944456993148866
>> 0.9977505546102384
>> 1.0340802667750915
>> 1.0713852057181472
>> 1.1096817412523137
>> 1.1489847692011224
>> 1.4069516500592667
>> 12.709029785778215
```

As you can see, the cost is is normally ≈ 1; however at the end it peaks to ≈ 12. The cost is the squared difference between the actual and predicted y co-ordinate of a point. This cost is the last one output and thus is in relation to the point $(100, 10101)$. Since this point is so far away from the rest of the data (that it could even be statistically classed as an outlier) the deviation of the hypothesis function is expected to be incorrect by this amount ($\sqrt{12}$).

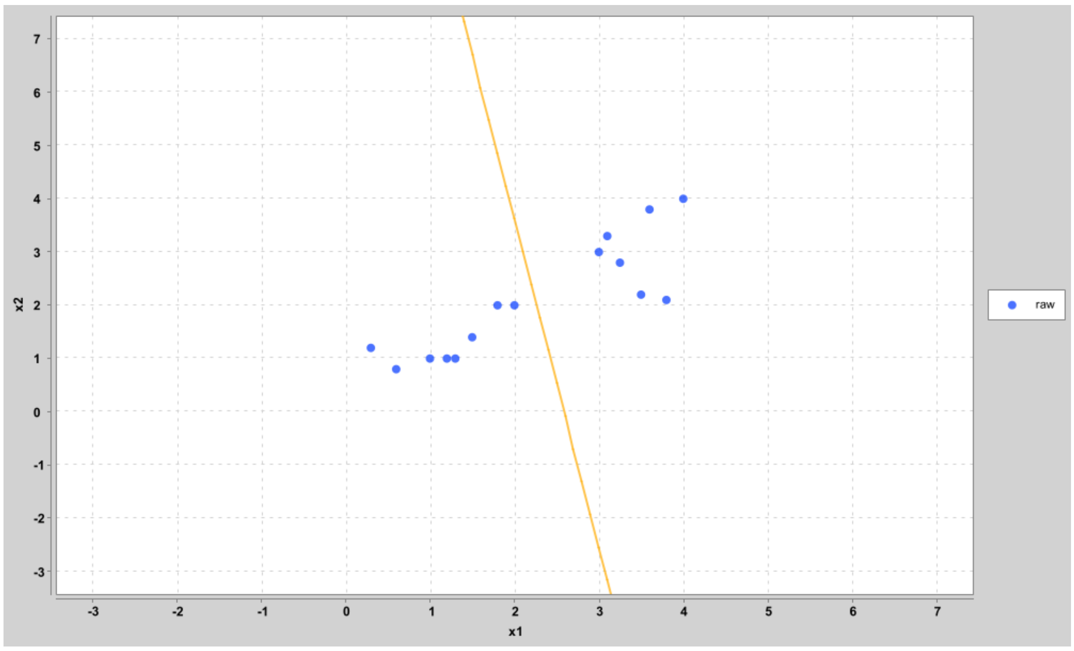
[1]solution1.java

Question Two:

Multivariate Logistic Regression with inital values of w_0, w_1 and w_2 set to 0 and α set to 0.1 over 100 epochs. My code[2] yields the following hypothesis function and the associated weights:

```
>> h(x) = (((-2.634179597653396) * x) - -6.816144929215793) / -6.816144929215793
>> w2 = -6.816144929215793
>> w1 = 2.634179597653396
>> w0 = -6.816144929215793
```

After plotting this function (using xchart) we can see the decision boundary function fits the separate classes of points quite well.



My code also outputs how much the cost changes by each time it is incremented; I have shown the last few lines below:

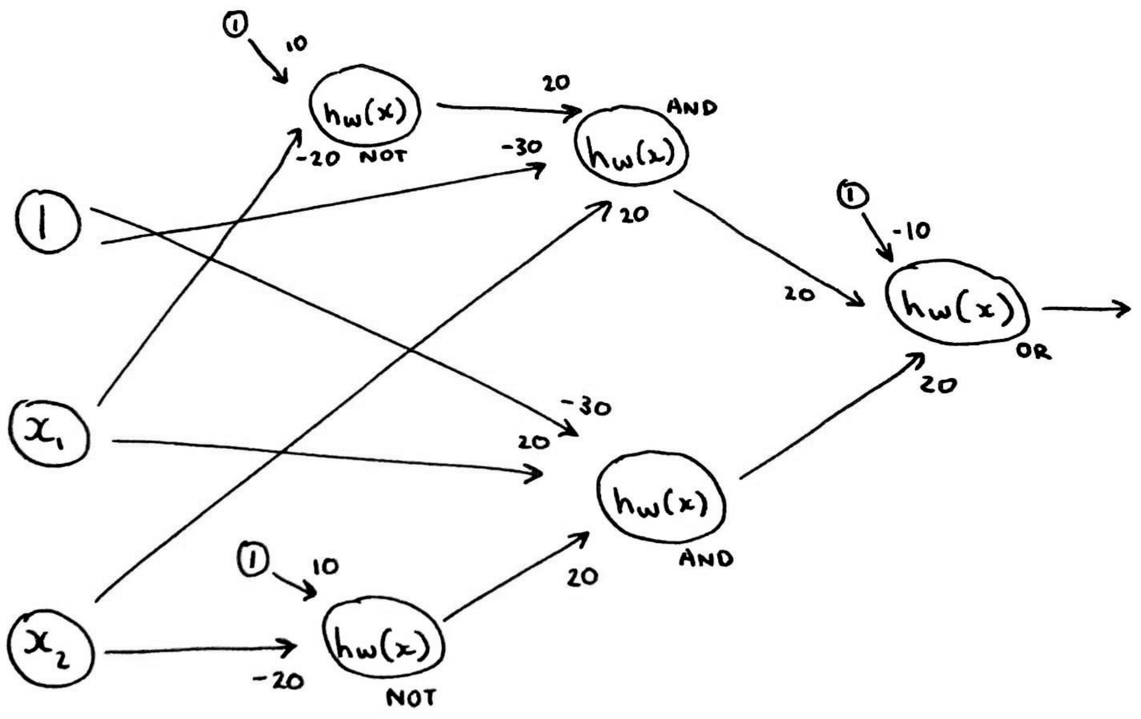
```
>> ...
>> -4.199751525602771
>> -2.7425767118276188
>> -3.706825493597943
>> 0.1884845652501731
>> 0.008095068678063375
>> 0.09704676876513688
>> 0.018663634040316904
>> 0.020720688977319088
>> 0.04128209235201255
>> 0.05719333010465827
```

As you can see the cost is strictly decreasing after each iteration. This means that my decision boundry function is converging to an accurate result.

[2]solution2.java

Question 3:

Neural Network capable of performing the XNAND[3] operation on two variables x_1 and x_2:



x_1	x_2	z	$g(z)$
0	0	-10	0
0	1	20	1
1	0	20	1
1	1	-10	0

[3] $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$

solution1.java

```
public class solution1 {
    public static void main(String[] args) {
        //init data
        double[][] data = {{-100, 9901},
                {-10, 91}, {-3, 7}, {0, 1}, {1, 3},
                {2, 7}, {3, 13}, {4, 21}, {10, 111}, {100, 10101}};
        // Number of iterations we want to run through the algorithm
        final int epochs = 10000;
        // We want to predict h(x) = (w2 * x^2) + (w1 * x) + w0
        double w1 = 0;
        double w0 = 0;
        double w2 = 0;
        // Learning rate
        double alpha = .00000001;
        // Main Gradient Descent Function for Non-Linear Regression
        for (int i = 0; i < epochs; i++) {
            //initial cost declaration
            double cost = 0;
            //iterating through data
            for (int j = 0; j < data.length; j++) {
                //getting x,y co-ords
                double x_j = data[j][0];
                double y_j = data[j][1];
                //calculating value from quadratic h(x)
                double prediction = (w1 * x_j) + w0 + (w2 * Math.pow(x_j, 2));
                // cost += (y_j - h(x))^2
                cost += (y_j - prediction) * (y_j - prediction);
                //cost increment output
                System.out.println((y_j - prediction) * (y_j - prediction));
                // Update the parameters for our equation.
                w1 += alpha * (y_j - prediction) * x_j;
                w2 += alpha * (y_j - prediction) * Math.pow(x_j, 2);
                w0 += alpha * (y_j - prediction);

            }
        }
        System.out.println("h(x) = (" + w2 + " * x^2) + (" + w1 + " * x) + " + w0);
    }
}
```

solution2.java

```java
public class solution2 {
    public static void main(String[] args) {
        //init data
        double[][] data = {{1, 1, 0}, {2, 2, 0}, {0.3, 1.2, 0},
                {0.6, 0.8, 0}, {1.2, 1, 0}, {1.3, 1, 0}, {1.8, 2, 0},
                {1.5, 1.4, 0}, {3, 3, 1}, {4, 4, 1}, {3.1, 3.3, 1}, {3.6, 3.8, 1},
                {3.8, 2.1, 1}, {3.5, 2.2, 1}, {3.25, 2.8, 1}};
        // Number of iterations we want to run through the algorithm
        final int epochs = 100;
        // We want to predict h(x) = (w2 * x2) + (w1 * x1) + w0
        double w1 = 0;
        double w0 = 0;
        double w2 = 0;
        // Learning rate
        double alpha = .1;
        // Main  Function for Logistic Regression
        for (int i = 0; i < epochs; i++) {
            //initial cost declaration
            double cost = 0;
            //iterating through data
            for (int j = 0; j < data.length; j++) {
                //getting x1,x2,y data
                double x_1 = data[j][0];
                double x_2 = data[j][1];
                double y = data[j][2];
                //calculating value from h(x)
                double prediction = sigmoid((w1 * x_1) + w0 + (w2 * x_2));
                // cost based on class
                if (y == 1) {
                    cost += -(Math.log(prediction));
                    //cost increment output
                    System.out.println(-(Math.log(prediction)));
                }
                if (y == 0) {
                    cost += -(1 - Math.log(prediction));
                    //cost increment output
                    System.out.println(-(1 - Math.log(prediction)));
                }
                // Update the parameters for our equation.
                w1 += alpha * (y - prediction) * x_1;
                w2 += alpha * (y - prediction) * x_2;
                w0 += alpha * (y - prediction);
            }
        }
        System.out.println("w2 = " + w0);
        System.out.println("w1 = " + w1);
        System.out.println("w0 = " + w0);
    }
    static double sigmoid(double x) {
        return 1 / (1 + Math.pow(Math.E, -x));
    }
}
```