COMP2039 Artificial Intelligence Coursework

Edward Seabrook
Electronics and Computer Science
University of Southampton

April 11, 2012

Abstract

The problem of predicting the secondary structure of proteins is a difficult one. Traditional statistical methods don't tend to produce good results. In their 1987 paper Qian and Sejnowski showed that artificial neural networks can be used to gain more accurate predictions than previous methods have been able to give. This study shows the usefulness of perceptrons for solving this, and other problems, confirming the findings of the aforementioned paper. This report also explores state of the art online structure prediction services, that are able to outperform the neural networks used in the study by Qian et al.

1 Introduction

In Molecular Biology, a protein is a biochemical compound that contains one or more polypeptide chain. The primary structure of a protein is defined as the sequence of amino acids that make up the polypeptide chain. There are twenty amino acids that can form proteins, these are each represented by a different letter of the alphabet. The secondary structure, the one we are concerned with, refers to the general local structure of the protein at some amino acid, and is determined not only by the given amino acid but also by those that surround it. Secondary structures can be placed into one of three classes: α-helix, β-sheet and coil (anything that is not α or β). The task of predicting the secondary structure given only the primary structure is relatively difficult, traditional statistical methods rarely get above 50% success rates.

Artificial neural networks are a computational model that is based on the structure of the brain. They consist of one or more artificial neuron; each neuron has a set of inputs, a set of weights and a single output. A neural network can be "trained" on some set of inputs and expected outputs such that the output on unseen data will match the patterns contained in the training data. These networks can be used to model complex functions and discover high dimensional patterns. It is possible to apply a threshold to the output of the neurons and produce a binary classifier.

It has been shown (by Qian et al.) that binary classifying artificial neural networks can be used to predict the secondary structure of proteins with a noticeable improvement in success rates over traditional statistical methods. This study aims to validate these claims.

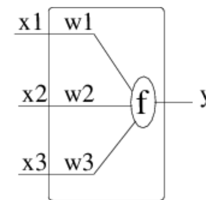


Figure 1: A conceptual model of a Perceptron

2 Method

2.1 Artificial Neural Networks

As previously mentioned an artificial neural network is a collection of artificial neurons that can be used to model some function; they are capable of modelling functions that contain patterns only at very high numbers of dimensions.

2.1.1 Perceptron

A perceptron is a the simplest kind of artificial neural network. It contains only one layer, and can be modelled as though it were only one neuron. Perceptrons are generally implemented as binary classifiers producing a result of one of two values (sigmoid perceptrons that produce a results scaled between zero and one also exist), this means that they will return one of two values, based on some internal threshold.

In this study perceptrons were implemented to

solve the decision problems: "is an α-helix" and "is a β-sheet". Outputs from these perceptrons were then combined to determine the secondary structure of the protein.

The outputs of the perceptrons were generated using the function:

$$f(x) = \begin{cases} 1 & \text{if } x \cdot w + b > 0 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Where x is a vector of inputs, w is a vector of weights and b is a bias. x is of equal length to w.

The perceptron implementation was designed to be as flexible and reusable as possible. The number of inputs, the bias learning rate and the number of iterations could all be altered depending on the application. This allows to the perceptron to be trained and tested on a wide variety of problems.

2.1.2 Multilayer Neural Networks

Multilayer Neural networks consist of many layers of nodes. The input layer receives input and the output layer returns values to the user. There are also hidden layers where the nodes receive inputs from other layers, and return values to further nodes in other layers. Multilayer neural networks are sensitive to patterns of higher order than perceptrons.

The encog Java neural network library was used to implement a multilayer neural network. To allow the tests written for the perceptron to run on the encog implementation, an adaptor class was written to provide a consistent interface to the two implementations.

2.1.3 Training

Neural networks require training before they can be used to predict anything. The aim of training is to obtain a set of weights that produce the expected corresponding results when inputs are passed into Equation 1. For each element of the training set, the elements must be adjusted to ensure the correct answer will be returned.

The training algorithm for a multilayer neural network is very complex and is beyond the scope of this project. The perceptron training algorithm is far simpler:

1. Begin by selecting some random values for each of the weights w_i. The bias b is set to some predefined constant.
2. For each input vector x_i of the training set D:
   (a) Calculate the actual output y_j of the perceptron using equation 1.
   (b) If y_j is:
      i. equal to the expected result d_j, do nothing.
      ii. larger than d_j, decrease w.
      iii. smaller than d_j, increase w.

Step two leads to the formula:

$$w_i(t+1) = w_i(t) + \alpha(d_j - y_j(t))x_{ij} \tag{2}$$

Where α is the learning rate constant.

We call an element of the dataset, where the actual output is not equal to the expected output, an error case. Step 2 is repeated until there are less than a certain threshold of error cases in the data set, or some predetermined limit for the number of iterations has been reached.

2.2 Dataset

The data set used in this study is published in the UCI Machine Learning Repository [1]. It is comprised of a list of the primary structures of many proteins and their corresponding secondary structures.

Initially the data set is very imbalanced. The distribution of α-helices, for example, is far lower than that of not α-helices. A more distributed subset of the training set needs to be created to train the neural network with - if this was not done, the neural network would essentially learn that α-helices never occur.

The data set fed into the neural networks also needs to take into account the fact that each secondary structure is affected not only by it's corresponding primary structure but also those surrounding it.

The perceptron implementation took care of these problems by first reading in the sequence of amino acids stored in the text file. Then for each entry it creates an AminoAcid object that stores the primary and secondary structure. Each AminoAcid object is stored inside a list containing class called a Protein. The Proteins are then stored inside a List. This object structure was very flexible and allowed the representation to be output in many different formats very easily.

With this abstract representation of the data, an intermediate structure is created, a three dimensional array of doubles. For each AminoAcid a "window" is stored - the window contains a central AminoAcid and then two (or any other number) either side. Where the window runs off the ends of

the protein, blanks are stored. Each AminoAcid is binary coded into a twenty one index array where all the values are zero except for the one representing the primary structure who's value is one. The expected output is stored as an array of length three in the second position of the second dimension, and is coded in a similar fashion. Each of the windows are stored in a separate index of the third dimension of the array.

The array might look something like:

```
{{{0,0,..,1,0}, {0, 0, 1}},
 {{1,0,..,0,0}, {0, 1, 0}},
 ..,
 {{0,0,..,0,1}, {1, 0, 0}}}
```

Once this array has been obtained, a set number of samples are picked at random, ensuring that the number of samples picked from each class (e.g. α-helix or not α-helix) are roughly equal; This is done by generating a random number and only using that selection if the random number is on the correct side of the required probability.

2.3 Measuring Performance

The two main measures of the success of a secondary structure prediction model are the correlation coefficients C_x, and the percentage of the predictions that are correct Q_3.[2]

The correlation coefficients can be calculated using the following formula:

$$C_x = \frac{(p_x n_x) + (u_x o_x}{\sqrt{(n_x + u_x)(n_x + o_x)(p_x + u_x)(p_x + o_x}} \quad (3)$$

Where p_x is the number of positively predicted that were correct, n_x is the number of correct negative predictions, u_x is the under predictions, and o_x is over predictions.

The percentage of correct predictions (or success rate) for the secondary structure problem can be calculated as below. This method can of course be generalised to other problems:

$$Q_3 = \frac{P_\alpha + P_\beta + P_{coil}}{N} \quad (4)$$

where P_x is the number of correctly predicted windows and N is the total number of windows.

For the perceptron, where there is only one output, the following adapted formula can be used, where $P_{!\alpha}$ is the percentage of correctly guessed non α-helicies:

$$Q_3 = \frac{P_\alpha + P_{!\alpha}}{N} \quad (5)$$

Table 1: Weights for OR Gate

Iteration	w_0	w_1	bias
0	0	0	1
1	1	1	-1

A similar formula can also be used for β-sheets. However, the correlation coefficient tends to be a better indicator of how good the model is for secondary structures.

A more qualitative view of the performance and other properties of the neural networks can be obtained by plotting graphs of various aspects of the data. In this study Gnuplot was used to produce the graphs from data written to stdout by the Java programs.

3 Results

3.1 Toy Problems

The problem of predicting the Secondary structure of a protein is a very high dimensional problem, even with only one amino acid, there are still twenty inputs. Due to the limitations of space and time, it is very difficult to plot a graph of more than three dimensions. Synthetic data sets that could be easily verified to test that the perceptron implementation was correct were produced to overcome this issue. For a perceptron to be able to learn a problem, that problem must be linearly separable, this means that there exists some place or hyperplane that can intersect the two classes. XOR is an example of a problem that is not linearly separable.

3.1.1 OR Gate

The first synthetic data set that was generated was an OR gate, the perceptron was trained with the same four input vectors until it generated the correct answer for each of them first time round.

Because the OR Gate is such as simple problem, starting with all the weights on zero, the perceptron was able to produce the correct weights after just one iteration. This can be seen from the weights of each iteration shown in Table 1.

Once trained the perceptron was able to produce a 100% success rate, predicting correctly all four possible inputs. It should be noted that the training set for this perceptron was the same as its test set.

Figure 2: Graph of random points split into two classes by the line 0.73x + 0.34

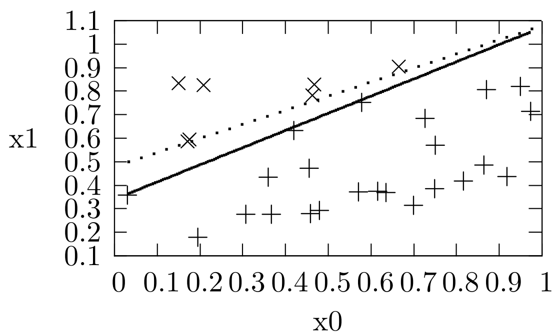
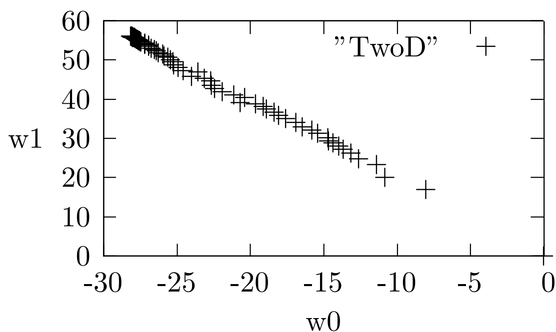


Figure 3: Graph of Weights for Linear Function



3.1.2 Linear function

A second, slightly less simple, dataset was generated at random, using a seed to ensure repeatability, that complied with the formula for a straight line:

$$x_1 = mx_0 + c \quad (6)$$

As can be seen in Figure 2, random points were classified depending on which side of the solid line they fall. On the graph, the crosses represent one class and the pluses the other. The perceptron is represented by the dotted line, it is not identical to the expected line but nevertheless it is able to correctly classify all the points on the graph.

This perceptron took many more iterations to produce acceptable weights than the OR gate perceptron. Figure 3 shows weights of the perceptron when $m = 2$ and $c = 0.7$. The clustering on the graph shows that as the weights approached their actual value, approximately 58 and -28, the difference in weights after each iteration decreased.

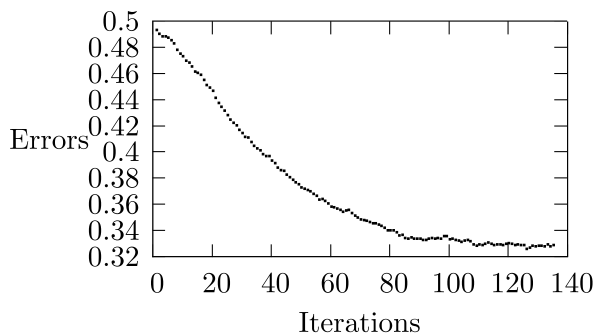
Using 100 different random sets of values for m and c, each over a test set of 10,000 pairs, only 15 perceptrons made any incorrect predictions, of these the average success rate was still 99.835% . It is worth noting that there was an iteration limit of 1001 on the training procedure, and all of the incorrect results had reached this limit. Given unbounded time and space, it is likely that all of the perceptrons would have accepted their entire test set. The overall average success rate was 99.975%.

Table 2: Iterations and Success Rates of a Perceptron Modelling the Function $x_1 = mx_0 + c$

m	c	Iterations	Success Rate
0.73	0.34	229	100.0%
0.17	0.71	58	100.0%
0.93	0.56	236	100.0%
0.21	0.37	229	100.0%
0.33	0.79	64	100.0%
0.67	1	1	100.0%
0.45	0.42	198	100.0%
0.04	0.98	1001	99.6%
0.1	0.76	822	100.0%
0.24	0.86	24	100.0%

Table 2 shows a small sample of the iterations and success rates for random m and c values, the actual values of m and c have been rounded.

Figure 4: Graph of the proportion of errors that occurred on each iteration for a perceptron



3.2 Predicting Secondary Structures

3.2.1 Perceptron

The same implementation of a perceptron was used to answer the decision problem "is an α-helix". A window of five amino acids was used as the input to the perceptron - this gave a total of 105 inputs to the perceptron.

With a learning rate of 0.0001 it took 140 iterations to reach a minimum proportion of errors at 77% success. The errors given on the graph is the number of error cases divided by the total number of inputs. This can be seen on the graph in Figure 4 . After this point the values remained stable, fluctuating around a point slightly higher than this minima.

At this number of iterations with a random balanced training set, and tested on the complete test

Figure 5: Graph of the success rare against window size for a multilayer neural network

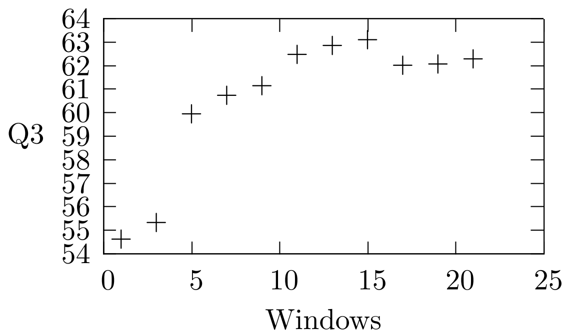
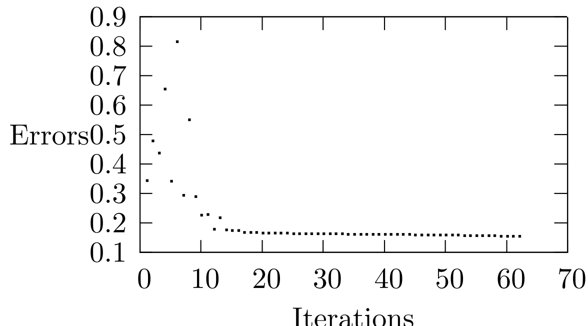


Figure 6: Graph of the proportion of errors that occurred on each iteration for a multilayer neural network



set, a value of 66.59% was obtained for $Q3$ and a value of 0.281 was obtained for C_α. The peceptron correctly guessed 1787 negatives(n) and 557 positives(p), incorrectly guessing 292 positives(u) and 884 negatives(o). These values were typical for several trial runs using different seeds to create the random numbers.

The perceptron was also altered to predict the presence of a β-sheet. The results were fairly similar to those obtained for α-helices. Values of $Q3 \approx 65\%$ and $C_\beta \approx 0.253$ were typical.

These two perceptrons were then assembled into one by creating a composite class that offered the same interface as the normal perceptron, but instead returned an array of three values, representing whether the structure was an α-helix, a β-sheet or a coil. If both of the constituent perceptrons return true, one of the two inputs is chosen at random. This method was only able to obtain $Q_3 \approx 55\%$, which is not as high as in (Qian)[2]. Correlelation coeffcients were $C_\alpha \approx 0.264$, $C_\beta \approx 0.221$ and $C_{coil} \approx 0.286$.

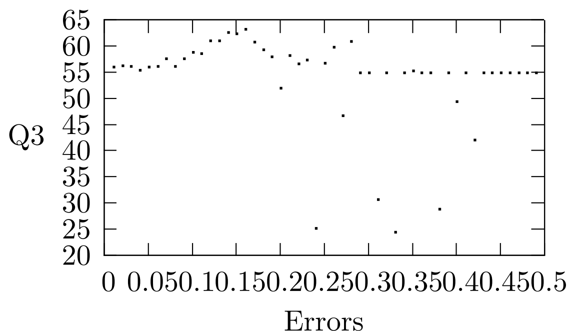
3.2.2 Multilayer neural network

To implement a Multilayer neural network the encog java library was used. [3] Encog offers a huge choice of classes for deploying a neural network; the BasicNetwork class was chosen as this is the most similar to the network used by (Qian). SigmoidActivation was used as the model for the network and ResilliantPropogation was used for training. An adaptor class was written to allow the tests written for the perceptron to be reused.

Initially a balanced training set was used, however this didn't produce brilliant results: a typical values was $Q_3 \approx 54$ and lower.

The training set was extended to the entire training data as in (Qian)[2], this produced results that were more similar to those in (Qian). The window size was also increased from 5 to 13; as

Figure 7: Graph of the success rare against error threshold for a multilayer neural network



(Qian) showed, Figure 5 confirms that the optimal number of amino acids per window is 13. Typical results of $Q_3 \approx 62.8\%$, $C_\alpha \approx 0.309$, $C_\beta \approx 0.295$ and $C_{coil} \approx 0.376$ were obtained.

As can be seen in Figure 6, the training seems to start off fairly sporadically, but quickly settles down tending towards zero at a decreasing rate. Strangely it seems that number of errors that occur during training are not directly proportional to the success rate, peaking at around 0.15. This can be seen in Figure 7

4 Discussion

4.1 Comparison with Qian et Sejnowski

Using a perceptron with a balanced data set, predicting just α-helix and not α-helix, values for success rate (Q_3) higher than in (Qian) were obtained. This is probably the case because the perceptron is not distinguishing between two of the classes (β-sheet and coil). When two perceptrons were assembled together to classify in one of the three classes, far lower results were obtained. An explanation for this is that: a) The window size was smaller b) there is no hidden layer, so any second order

Table 3: Results of Online Secondary Structure Predictors

Service	Q_3	C_α	C_β	C_{coil}
Qian [2]	64.3%	0.41	0.31	0.41
Jpred [5]	76%	0.80	0.47	0.58
PSIPRED [6]	78%	0.80	0.52	0.62
Porter [7]	70%	0.74	0.46	0.52
JUfo [8]	69%	0.77	0.43	0.50
Scratch [9]	71%	0.66	0.54	0.49

patterns in the data set cannot be identified.

The multilayer network (encog) produced results that were far more similar to those of (Qian), this is mainly because the structure of the network was very similar, and the parameters used when constructing and configuring the objects were based on those used by (Qian). The fact that the results were so similar and the trends which were observed by varying the independent variables shows that (Qians) results were reproducible and therefore valid.

4.2 Comparison with Online Services

The string representing the amino acids of the first protein in the test set was fed into several online services. The outputs were then parsed and the various statistics generated for them. These results can be seen in Table 3.

The results obtained from these web based services tended to be much higher than anything that was produced by the predictor in this study, or the results reported by (Qian). There could be many reasons for this, including: The testing data used was very small, and could have been an anomalous case, the modern services employ better algorithms than existed when (Qian) was published, the modern services use a larger data set, or the protein may occur in (or be homologous with one in) the training set employed by these services.

The web based services all took a very long time to return answers, averaging around ten minutes. This could also be for the same reasons as above, or for other reasons such as competition on the server causing delays before processing.

4.3 Conclusion

This study has shown that neural networks can be used to solve very simple problems with a very high level of accuracy, and more complex problems with a lower level of accuracy.

Neural networks are clearly a better way of predicting secondary structure than traditional methods, but still are not very reliable. Modern systems that use other methods to obtain a higher success rate are clearly more desirable for use in scientific and medical settings, where a success rate of 63% wouldn't be acceptable - although it is certain that there are many applications where success rates of 78% are not high enough either.

References

[1] UCI Machine Learning Repository, http://archive.ics.uci.edu/ml/.

[2] Qian and Sejnowski, Predicting the Secondary Structure of Globular Proteins Using Neural Network Models. Academic Press Limited, Journal of Molecular Biology (1988) 202, 865-884

[3] Encog, Heaton Research, http://www.heatonresearch.com/encog

[4] The wikipedia Entry for Perceptrons, http://en.wikipedia.org/wiki/Perceptron

[5] Jpred 3, University of Dundee, http://www.compbio.dundee.ac.uk/www-jpred/

[6] PSIPRED Protein Structure Prediction Server, University College London, http://bioinf.cs.ucl.ac.uk/psipred/

[7] Porter University College Dublin http://distill.ucd.ie/porter/

[8] JUfo Vanderbilt University http://meilerlab.org/index.php/servers/show?s_id=5

[9] Scratch Information and Computer Sciences University of Califonia Irvine http://scratch.proteomics.ics.uci.edu/

Appendices

The following listings are the code used to produce the results seen in this report. The encog Java library is required to run this code. The code provided runs silently, to produce an output, "System.out.println()" statements must be added in the correct places. The Main methods in the various classes are included as examples and should be altered to perform the desired functionality.

A Amino Acid Class

```
import java.util.HashMap;


/**
 *  Represents an amino acid.
 */
public class AminoAcid{
    Character letter;
    boolean isAlpha;
    boolean isBeta;
    boolean isCoil;

    static boolean letterMapInitialized = false;
    static boolean inverseLetterMapInitialized = false;
    static HashMap<Character, Integer> letterMap;
    static HashMap<Integer, Character> inverseLetterMap;

    /**
     * The default constructor for an amino acid
     * @param letter The letter that this amino acid is represented by
     * @param secondaryStructure The secondary structure associated
     * with this amino acid
     */
    public AminoAcid(Character letter, char secondaryStructure){
        this.letter = letter;

        initializeLetterMap();
        initializeInverseLetterMap();

        isAlpha = false;
        isBeta = false;
        isCoil = false;

        switch(secondaryStructure){
            case 'h': isAlpha = true; break;
            case 'e': isBeta = true; break;
            case '_': isCoil = true; break;
        }
    }

    /**
     * Sets up the letter map
     */
    static private void initializeLetterMap(){
```

```java
        if(letterMapInitialized) return;

        letterMap = new HashMap<Character, Integer>();

        letterMap.put('G', 0);
        letterMap.put('P', 1);
        letterMap.put('A', 2);
        letterMap.put('V', 3);
        letterMap.put('L', 4);
        letterMap.put('I', 5);
        letterMap.put('M', 6);
        letterMap.put('C', 7);
        letterMap.put('F', 8);
        letterMap.put('Y', 9);
        letterMap.put('W', 10);
        letterMap.put('H', 11);
        letterMap.put('K', 12);
        letterMap.put('R', 13);
        letterMap.put('Q', 14);
        letterMap.put('N', 15);
        letterMap.put('E', 16);
        letterMap.put('D', 17);
        letterMap.put('S', 18);
        letterMap.put('T', 19);
    }

    /**
     * Sets up the inverse letter map
     */
    static private void initializeInverseLetterMap(){
        if(inverseLetterMapInitialized) return;

        inverseLetterMap = new HashMap<Integer, Character>();

        inverseLetterMap.put(0, 'G');
        inverseLetterMap.put(1, 'P');
        inverseLetterMap.put(2, 'A');
        inverseLetterMap.put(3, 'V');
        inverseLetterMap.put(4, 'L');
        inverseLetterMap.put(5, 'I');
        inverseLetterMap.put(6, 'M');
        inverseLetterMap.put(7, 'C');
        inverseLetterMap.put(8, 'F');
        inverseLetterMap.put(9, 'Y');
        inverseLetterMap.put(10, 'W');
        inverseLetterMap.put(11, 'H');
        inverseLetterMap.put(12, 'K');
        inverseLetterMap.put(13, 'R');
        inverseLetterMap.put(14, 'Q');
        inverseLetterMap.put(15, 'N');
        inverseLetterMap.put(16, 'E');
        inverseLetterMap.put(17, 'D');
        inverseLetterMap.put(18, 'S');
        inverseLetterMap.put(19, 'T');
```

```
    }

    /**
     * @return The letter associated with this amino acid
     */
    public Character getLetter(){
        return this.letter;
    }

    /**
     * Converts an array index to an offset
     * @param num The array offset
     * @return The character representation
     */
    static public char getLetterFromNumber(int num){
        return inverseLetterMap.get(num);
    }

    /**
     * @return The array offset needed to represent this amino acid
     */
    public int getOffset(){
        return letterMap.get(this.getLetter());
    }

    /**
     * @return true if the secondary structure is alpha helix
     */
    public boolean getIsAlpha(){
        return this.isAlpha;
    }

    /**
     * @return true if the secondary structure is beta sheet
     */
    public boolean getIsBeta(){
        return this.isBeta;
    }

    /**
     * @return true if the secondary structure is coil
     * @return
     */
    public boolean getIsCoil(){
        return this.isCoil;
    }

}
```

B Multi Perceptron Class

```
import java.util.Random;

/**
```

```java
 * This class uses two perceptrons to predict the secondary
 * structure of a protein.
 */
public class MultiPerceptron implements PerceptronInterface{

    Perceptron hPerceptron, ePerceptron;
    static final double POSITIVE = 1, NEGATIVE = 0;

    /**
     * Defualt constructor for MultiPerceptron
     * @param learningRate A user defined rate of learning
     * @param bias The bias to be used in the perceptrons
     * @param maxIterations A limit to the iterations to perform
     * @param weightRange The range in which the random weights will fall
     * @param noOfInputs The number of inputs to the perceptrons
     */
    public MultiPerceptron(double learningRate, double bias,
            int maxIterations, double weightRange, int noOfInputs){
        hPerceptron = new Perceptron(learningRate, bias, maxIterations,
                weightRange, noOfInputs);
        ePerceptron = new Perceptron(learningRate, bias, maxIterations,
                weightRange, noOfInputs);
        ePerceptron.setExpectedResultIndex(1);
    }

    /**
     * Trains the two contained perceptrons based on the data set
     * @param dataSet The complete (unbalanced) data set to be trained on
     * @param expectedResults The expected results
     */
    public void learnFromData(double[][] dataSet, double[][] expectedResults) {
        //Create a balanced data set for the alpha predictor
        double[][][] bds =
                (new SecondaryStructurePredictor('h')).createBalancedDataSet(
                        dataSet, expectedResults, 7200);
        hPerceptron.learnFromData(bds[0], bds[1]);
        //Create a balanced data set for the beta predictor
        bds = (new SecondaryStructurePredictor('e')).createBalancedDataSet(
                dataSet, expectedResults, 7200);
        ePerceptron.learnFromData(bds[0], bds[1]);
    }

    /**
     * Predicts a secondary structure using the two perceptrons
     * @param input The input to predict a result for
     */
    public double[] predict(double[] input) {
        //IF they are both negative, set coil to true
        double coil;
        if((hPerceptron.predict(input)[0] == NEGATIVE) &&
                (ePerceptron.predict(input)[1] == NEGATIVE)){
            coil = POSITIVE;
        }
        else coil = NEGATIVE;
```

```
        double[] result = {hPerceptron.predict(input)[0] ,
                ePerceptron.predict(input)[1], coil};

        //If they are both negative, pick just one
        Random r = new Random(32);
        if((hPerceptron.predict(input)[0] == POSITIVE) &&
                (ePerceptron.predict(input)[1] == POSITIVE)) {
            result[r.nextInt(2)] = NEGATIVE;
        }
        return result;
    }

}
```

C Neural Network Class

```
import java.util.Arrays;
import org.encog.ml.data.MLData;
import org.encog.ml.data.MLDataSet;
import org.encog.ml.data.basic.BasicMLData;
import org.encog.ml.data.basic.BasicMLDataSet;
import org.encog.ml.train.MLTrain;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.training.propagation.resilient.ResilientPropagatio

/**
 * Adaptor from Perceptron interface to Encog
 */
public class NeuralNet implements PerceptronInterface{
    BasicNetwork network;
    int outputs;
    double errors;

    public static int SEED = 42;
    static final double POSITIVE = 1, NEGATIVE = 0;

    /**
     * Default constructor for Neural Net class
     * @param inputs Number of inputs
     * @param outputs Number of outputs
     * @param hiddenLayers Number of units in hidden layer
     * @param errors The acceptance threshold for iterations
     */
    public NeuralNet(int inputs, int outputs, int hiddenLayers, double errors){
        network = new BasicNetwork();
        network.addLayer(new BasicLayer(inputs));
        network.addLayer(new BasicLayer(hiddenLayers));
        network.addLayer(new BasicLayer(outputs));
        network.getStructure().finalizeStructure();
        //Unfortunately introduces a random element
        network.reset();

        this.outputs = outputs;
```

```
        this.errors = errors;
    }

    /**
     * Teaches the neural network based on data set
     * @param dataSet The data set to learn from
     * @param expectedResults The corresponding expected results
     */
    public void learnFromData(double[][] dataSet, double[][] expectedResults) {
        MLDataSet trainingSet = new BasicMLDataSet(dataSet, expectedResults);
        final MLTrain train = new ResilientPropagation(network, trainingSet);
        do{
            train.iteration();
        }while((train.getIteration() < 1000) && (train.getError() > errors));
    }

    /**
     * Guesses The secondary structure for the given input
     * @param input The data to guess for
     * @return Some guesses
     */
    public double[] predict(double[] input) {
        MLData test = new BasicMLData(input);

        //Takes care of any perceptron style problems
        if (outputs == 1){
            double[] result = {1};
            if(network.compute(test).getData()[0] > 0.5) result[0] = POSITIVE;
            else result[0] = NEGATIVE;
            return result;
        }

        //Takes care of the rest
        double currentHighest = network.compute(test).getData()[0];
        int currentHighestIndex = 0;
        for (int i = 0; i < outputs; i++){
            if (network.compute(test).getData()[i] > currentHighest){
                currentHighest = network.compute(test).getData()[i];
                currentHighestIndex = i;
            }
        }
        double[] result = new double[outputs];
        Arrays.fill(result, NEGATIVE);
        result[currentHighestIndex] = POSITIVE;
        return result;
    }
}
```

D OR Gate Perceptron Test Class

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
```

```java
/**
 * This class encapsulates a test harness for an
 * OR gate perceptron
 */
public class ORGatePerceptronTest implements Test{

    static final int SEED = 42;
    static final boolean FAIR_DISTRIBUTION = true;
    static final double POSITIVE = 1, NEGATIVE = 0;

    /**
     * Example of a Main method for this class
     */
    public static void main(String[] args){
        Random r = new Random(SEED);
        Test t = new ORGatePerceptronTest();
        performTest(t , 30);
    }

    /**
     * Perform some test
     * @param test The test to perform
     * @param noOfTests No of times to repeat the test
     */
    static void performTest(Test test, int noOfTests){

        Perceptron p = new Perceptron(0.5, 1, 1000, 0, 2);

        int tests = noOfTests;

        double[][][] dataSetAndExpectedResults = test.generateDataset(tests);
        double[][] dataSet = dataSetAndExpectedResults[0];
        double[][] expectedResults = dataSetAndExpectedResults[1];

        p.learnFromData(dataSet, expectedResults);

        int failCount = 0;
        double[] input = new double[2];

        dataSetAndExpectedResults = test.generateDataset(tests);
        dataSet = dataSetAndExpectedResults[0];
        expectedResults = dataSetAndExpectedResults[1];

        List<double[]> setA = new ArrayList<double[]>(),
                setB= new ArrayList<double[]>();

        for(int i = 0; i < dataSet.length; i++){
            input[0] = dataSet[i][0];
            input[1] = dataSet[i][1];
            if(p.predict(input)[0] != expectedResults[i][0]) failCount++;
            if(expectedResults[i][0] == 1) setA.add(dataSet[i]);
            else setB.add(dataSet[i]);
        }
    }
```

```java
    /**
     * Generate an OR gate data set
     * @param size Not needed
     * @return a data set and expected results for OR gate
     */
    public double[][][] generateDataset(int size){
        double[][] dataSet = {{0,0}, {0,1}, {1,0}, {1,1}};
        double[][] expectedResults =
            {{NEGATIVE}, {POSITIVE}, {POSITIVE}, {POSITIVE}};

        double[][][] result = {dataSet, expectedResults};
        return result;
    }
}
```

E Perceptron Class

```java
import java.util.Random;
import java.util.Arrays;

/**
 * Perceptron is class that implements a perceptron to solve discision problems
 * based on learning from a data set of known values.
 */
public class Perceptron implements PerceptronInterface{
    static final int SEED = 42;
    static final double POSITIVE = 1, NEGATIVE = 0;

    private double learningRate;
    private double[] weight;
    private double bias;
    private int maxIterations;
    private int expectedResultIndex = 0;

    /**
     * The default constructor for Perceptron
     * @param learnignRate A user specified learning rate constant
     * @param bias A user specified bias
     * @param maxIterations The maximum number of iterations that will be
     * performed for a member of the input set
     * @param noOfInputs The number of inputs this perceptron will take
     */
    public Perceptron(double learningRate, double bias,
            int maxIterations, double weightRange, int noOfInputs){
        this.learningRate = learningRate;
        this.bias = bias;
        this.maxIterations = maxIterations;
        this.weight = new double[noOfInputs];

        Arrays.fill(this.weight, 0);

        if (weightRange != 0){
            Random r = new Random(SEED);
```

```java
        for(int i = 0; i < weight.length; i++) {
            weight[i] = (r.nextDouble() * weightRange) - (weightRange / 2);
        }
    }
}

/**
 * Takes in a data set and learns from the data
 * @param dataSet A set of known mappings in the desired function.
 * Must be of the form [x1, x2, ... , xn, desired output]
 */
public void learnFromData(double[][] dataSet, double[][] expectedResults){
    int errorCount = 0;
    int iterations = 0;

    do{

        iterations++;
        errorCount = 0;
        int i =0;
        for(double[] data: dataSet){
            if (teach(data, (int)expectedResults[i][expectedResultIndex])!=0){
                errorCount++ ;
            }
            i++;
        }
    } while(errorCount > 1 && iterations <= maxIterations);
}

/**
 * Teaches the machine the given input
 * @param input The know input of the function
 * @param expectedResult The value the function should return when given input
 * @return error
 */
private int teach(double[] input, int expectedResult){
    double actualResult, resultDifference;
    int error = 1;

    actualResult = predict(input)[expectedResultIndex];
    resultDifference = expectedResult - actualResult;

    if (resultDifference == 0) return 0;

    bias = bias + learningRate * resultDifference * 1;

    for(int i = 0; i < input.length; i++){
        weight[i] = weight[i] + learningRate * resultDifference * input[i];
    }
    return error;
}

/**
 * Predicts the value of f for the given input based on what
```

```
     * it has been taught
     * @param input Input data for f
     * @return +1 or -1 dependent on what it has learned
     */
    public double[] predict(double[] input){
        double y = getBias();

        for(int i = 0; i < input.length; i++){
            y = y + weight[i] * input[i];
        }

        double[] positive = new double[weight.length];
        positive[expectedResultIndex] = POSITIVE;
        double[] negative = new double[weight.length];
        negative[expectedResultIndex] = NEGATIVE;


        if(y >= 0) return positive;
        else return negative;
    }

    /**
     * @param index The index of the width
     * @return The weight of the index
     */
    public double getWeight(int index){
        return weight[index];
    }

    /**
     * @return The Bias
     */
    public double getBias(){
        return bias;
    }
    /**
     * Sets the expected result index
     * Some perceptrons may not use 0 for their expected results
     * @param index The idex to use
     */
    public void setExpectedResultIndex(int index){
        expectedResultIndex = index;
    }
}
```

F Perceptron Interface

```
/**
 * The interface that must be implemented by any perceptron like device
 */
public interface PerceptronInterface {
    public void learnFromData(double[][] dataSet, double[][] expectedResults);
    public double[] predict(double[] input);
}
```

G Protein Class

```
import java.util.ArrayList;
import java.lang.IndexOutOfBoundsException;

/**
 * A list of Amino Acids, Represents a protein
 */
public class Protein{
    ArrayList<AminoAcid> aminoAcids = new ArrayList<AminoAcid>();

    /**
     * Add an amino acid to the protein
     * @param aminoAcid amino acid to add
     */
    public void add(AminoAcid aminoAcid){
        aminoAcids.add(aminoAcid);
    }

    /**
     * @return The number of amino acids in this protein
     */
    public int size(){
        return aminoAcids.size();
    }

    /**
     * @param index The index of the amino acid
     * @return The Amino Acid at that index
     * @throws IndexOutOfBoundsException
     */
    public AminoAcid get(int index) throws IndexOutOfBoundsException{
        return aminoAcids.get(index);
    }
}
```

H Secondary Structure Predictor Class

```
import java.util.Arrays;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Set;
import java.io.InputStreamReader;
import java.io.FileInputStream;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.FileNotFoundException;
import java.lang.IndexOutOfBoundsException;
import java.util.Random;

/**
 * Controls the prediction of secondary structures
 */
public class SecondaryStructurePredictor {
```

```java
static int WINDOW_SIZE = 13;
static final int SEED = 42;
static final double POSITIVE = 1, NEGATIVE = 0;
PerceptronInterface perceptron;
char structure;

/**
 * Example of how to use this crazy code
 */
public static void main(String[] args){
    double learningRate = 0.00001;
    double bias = 1;
    int maxIterations = 656;
    double weightRange = 0.6;
    int noOfInputs = WINDOW_SIZE * 21;
    SecondaryStructurePredictor sp;

    sp = new SecondaryStructurePredictor('h');
    sp.testPeceptron(new Perceptron(learningRate, bias,
            maxIterations, weightRange, noOfInputs));

    sp = new SecondaryStructurePredictor('e');
    sp.testPeceptron(new Perceptron(learningRate, bias,
            maxIterations, weightRange, noOfInputs));

    sp = new SecondaryStructurePredictor('_');
    sp.testPeceptron(new Perceptron(learningRate, bias,
            maxIterations, weightRange, noOfInputs));

    sp = new SecondaryStructurePredictor('*');
    sp.testPeceptron(new MultiPerceptron(learningRate, bias,
            maxIterations, weightRange, noOfInputs));

    sp = new SecondaryStructurePredictor('*');
    sp.testPeceptron(new NeuralNet(WINDOW_SIZE * 21, 3, 40, 0.15));
}

/**
 * Default constructor for Secondary Structure Predictor
 * @param structure The structure to look out for
 * Either 'h', 'e', '_' or '*'
 */
public SecondaryStructurePredictor(char structure){
    this.structure = structure;
}

/**
 * Run the tests
 * @param perceptron Perceptron to run tests on
 */
void testPeceptron(PerceptronInterface perceptron){
    this.perceptron = perceptron;
```

```
if(perceptron instanceof Perceptron){
    switch(structure){

    case 'e':
        ((Perceptron)perceptron).setExpectedResultIndex(1); break;

    case '_':
        ((Perceptron)perceptron).setExpectedResultIndex(2); break;

    }
}

try{
    //This will need changing to run on any other pc
    ArrayList<Protein> proteinList =
            getProteinsFromFile("protein-secondary-structure.train");

    //Count Amino Acids
    int noOfAminoAcids = 0;

    for(Protein protein: proteinList){
        noOfAminoAcids = noOfAminoAcids + protein.size();
    }

    double[][] totalDataSet = new double[noOfAminoAcids][0];
    double[][] totalExpectedResults = new double[noOfAminoAcids][3];

    int arrayIndex = 0;

    for(Protein protein : proteinList){
        double[][][] dataSetAndExpectedResults =
                generateDataSet(protein);
        for (int i = 0; i < dataSetAndExpectedResults[0].length; i++){
            totalDataSet[arrayIndex] = dataSetAndExpectedResults[0][i];
            totalExpectedResults[arrayIndex] =
                    dataSetAndExpectedResults[1][i];
            arrayIndex++;
        }
    }

    double[][][] balancedDataSetAndExpectedResults =
            createBalancedDataSet(totalDataSet,
                    totalExpectedResults, 7200);
    perceptron.learnFromData(balancedDataSetAndExpectedResults[0],
            balancedDataSetAndExpectedResults[1]);

    //This will need changing to run on any other pc
    proteinList =
            getProteinsFromFile("protein-secondary-structure.train");

    calculatePercentageCorrect(proteinList);
}
catch(FileNotFoundException fnfe){
    System.out.println("File not found");
```

```
            System.exit(1);
        }
        catch(IOException ioe){
            System.out.println("Reading file Failed");
            System.exit(2);
        }
    }

    /**
     * Calculates the average correlation coefficient for the given Proteins
     * @param proteinList A list of proteins
     * @param structure The structure to check for
     * @return Correlation coefficient
     */
    private double calculateAverageCorrelationCoefficient(
            ArrayList<Protein> proteinList, char structure){
        long p = 0; long n = 0;
        long u = 0; long o = 0;

        int s = 0;
        switch (structure){
        case 'h': s = 0; break;
        case 'e': s = 1; break;
        case '_': s = 2; break;
        }

        for(Protein protein : proteinList){
            double[][][] dataSetAndExpectedResults = generateDataSet(protein);
            double[][] ds = dataSetAndExpectedResults[0];

            for(int i = 0; i < protein.size(); i++){
                if ((getIsStructure(protein.get(i), structure)) &&
                        (perceptron.predict(ds[i])[s] == POSITIVE))
                    p++;
                else if(!(getIsStructure(protein.get(i), structure)) &&
                        (perceptron.predict(ds[i])[s] == NEGATIVE))
                    n++;
                else if((getIsStructure(protein.get(i), structure)) &&
                        (perceptron.predict(ds[i])[s] == NEGATIVE))
                    u++;
                else if(!(getIsStructure(protein.get(i), structure)) &&
                        (perceptron.predict(ds[i])[s] == POSITIVE))
                    o++;
            }
        }

        double numerator = (p * n) - (u * o);
        long denominatorSquared = (n + u) * (n + o) * (p + u) * (p + o);
        double denominator = Math.sqrt(denominatorSquared);
        return numerator / denominator;
    }

    /**
     * Calculates the success rate
```

```
 * @param proteinList A list of proteins
 * @return Percentage succeeded
 */
private double calculatePercentageCorrect(ArrayList<Protein> proteinList){
    int count = 0; int total = 0;

    int s = 0;
    switch (structure){
    case 'h': s = 0; break;
    case 'e': s = 1; break;
    case '_': s = 2; break;
    }

    Random r = new Random(36);

    for(Protein protein : proteinList){
        double[][][] dataSetAndExpectedResults = generateDataSet(protein);
        double[][] ds = dataSetAndExpectedResults[0];

        for(int i = 0; i < protein.size(); i++){
            if(structure != '*'){
                if ((getIsStructure(protein.get(i), structure)) &&
                        (perceptron.predict(ds[i])[s] == POSITIVE))
                    count++;
                else if(!(getIsStructure(protein.get(i), structure)) &&
                        (perceptron.predict(ds[i])[s] == NEGATIVE))
                    count++;
            }
            else if(structure == '*'){
                double h = perceptron.predict(ds[i])[0];
                double e = perceptron.predict(ds[i])[1];
                double c = perceptron.predict(ds[i])[2];

                if ((getIsStructure(protein.get(i), 'h')) &&
                        (h == POSITIVE))
                    count++;
                else if ((getIsStructure(protein.get(i), 'e')) &&
                        (e == POSITIVE) )
                    count++;
                else if ((getIsStructure(protein.get(i), '_')) &&
                        (c == POSITIVE))
                    count++;
            }
            total++;
        }
    }
    return (((double) count / total) * 100);
}

/**
 * Creates a balanced data set depending on the structure of the class
 * @param dataSet The unbalanced data set
 * @param expectedResults The corresponding correct results
 * @param size The size of the balanced data set
```

```java
 * @return Balanced data set [0] and expected result [1]
 */
double[][][] createBalancedDataSet(
        double[][] dataSet, double[][] expectedResults, int size){
    double[][] balancedDataSet = new double[size][0];
    double[][] balancedExpectedResults = new double[size][1];

    if (structure == '*') {
        double[][][] result = {dataSet, expectedResults};
        return result;
    }

    Random r = new Random(SEED);
    int j;
    double p;
    boolean found;
    int structureInt = 0;
    Set<Integer> s = new HashSet<Integer>();

    switch(structure){
    case 'h': structureInt = 0; break;
    case 'e': structureInt = 1; break;
    case '_': structureInt = 2; break;
    }

    int k =0;
    for(int i = 0; i < size; i++){
        found = false;
        p = r.nextDouble();
        while(!found){
            j = r.nextInt(dataSet.length);
            if(((p < 0.5) && (expectedResults[j][structureInt] == POSITIVE))
            || ((p >=0.5) && (expectedResults[j][structureInt] == NEGATIVE))){
                if(!s.contains(j)){
                    s.add(j);
                    balancedDataSet[i] = dataSet[j];
                    balancedExpectedResults[i] = expectedResults[j];
                    found = true;
                    if(p >= 0.5) k++;
                }
            }
        }
    }
    double[][][] result = {balancedDataSet, balancedExpectedResults};
    return result;
}

/**
 * Uses the object representation to produce a data set
 * @param protein The protein to generate a data set for
 * @return Data set [0] and expected result [1]
 */
double[][][] generateDataSet(Protein protein){
    final int windowSize = SecondaryStructurePredictor.WINDOW_SIZE;
```

```
        final int noOfAminoAcids = 20;
        final int emptySpace = 1;

        int arraySize = windowSize * (noOfAminoAcids + emptySpace);

        double[][] dataSet = new double[protein.size()][arraySize];
        double[][] expectedResults = new double[protein.size()][3];

        for(double[] d : dataSet){
            Arrays.fill(d, NEGATIVE);
        }

        for(int i = 0; i < protein.size(); i++){
            for (int j = -(windowSize - 1) / 2; j <= (windowSize - 1) / 2; j++){
                try{
                    dataSet[i][(j + (windowSize - 1) / 2) * 21 +
                                protein.get(i + j).getOffset()] = POSITIVE;
                }
                catch(IndexOutOfBoundsException ioobe){
                    dataSet[i][(j + (windowSize - 1) / 2) * 21 + 20] = POSITIVE;
                }
            }
            expectedResults[i][0] = getIsStructureNum(protein.get(i), 'h');
            expectedResults[i][1] = getIsStructureNum(protein.get(i), 'e');
            expectedResults[i][2] = getIsStructureNum(protein.get(i), '_');
        }

        double[][][] result = {dataSet, expectedResults};
        return result;
    }

    /**
     * @param aminoAcid The amino acid to find the structure of
     * @param structure The structure to compare with
     * @return true if that amino acid is that structure
     */
    private boolean getIsStructure(AminoAcid aminoAcid, char structure){
        if(getIsStructureNum(aminoAcid, structure) == 1.0) return true;
        else return false;
    }

    /**
     * @param aminoAcid The amino acid to find the structure of
     * @param structure The structure to compare with
     * @return POSITIVE if that amino acid is that structure otherwise NEGATIVE
     */
    private double getIsStructureNum(AminoAcid aminoAcid, char structure){
        if(structure == 'h')
            if(aminoAcid.getIsAlpha()) return POSITIVE;
            else return NEGATIVE;
        else if(structure =='e')
            if(aminoAcid.getIsBeta()) return POSITIVE;
            else return NEGATIVE;
        else if(structure =='_')
```

```
            if(aminoAcid.getIsCoil()) return POSITIVE;
            else return NEGATIVE;
        return 0;
    }

    /**
     * Reads in the proteins from the file
     * @param filename The filename to look in
     * @return List of proteins from that file
     * @throws IOException
     * @throws FileNotFoundException
     */
    ArrayList<Protein> getProteinsFromFile(String filename)
            throws IOException, FileNotFoundException{
        //Set up file stuff
        FileInputStream fis = new FileInputStream(filename);
        InputStreamReader in = new InputStreamReader(fis);
        BufferedReader br = new BufferedReader(in);

        ArrayList<Protein> proteinList = new ArrayList<Protein>();
        Protein protein = new Protein();

        boolean started = false;

        while(true){
            String line = br.readLine();
            if(line == null) break;
            else if(line.startsWith("<>")){
                //Start next protein
                started = true;
                protein = new Protein();
                proteinList.add(protein);
            }
            else if(line.startsWith("end")) started = false;
            else if(line.startsWith("<end>")) started = false;
            else if(started){
                protein.add(new AminoAcid(line.charAt(0), line.charAt(2)));
            }
        }
        return proteinList;
    }
}
```

I Test Interface

```
/**
 * The interface that tests must implement to be run as tests
 */
public interface Test {
    public double[][][] generateDataset(int size);
}
```

J Two Dimensional Perceptron Test Class

```java
import java.util.Random;
/**
 * This class encapsulates a test harness for a
 * Line guessing perceptron
 */
public class TwoDimensionalPerceptronTest implements Test{
    private double m = 2, c = 0.07;
    static final double POSITIVE = 1, NEGATIVE = 0;
    static final long SEED = 42;

    /**
     * Sets the gradient of the line
     * @param m Gradient of the line
     */
    public void setM(double m){
        this.m = m;
    }

    /**
     * Sets the offset of the line
     * @param c the Offset of the line
     */
    public void setC(double c){
        this.c = c;
    }

    /**
     * Generate a Line data set
     * @param size The number of points
     * @return a data set and expected results for Line
     */
    public double[][][] generateDataset(int size){
        double[][] dataSet = new double[size][2];
        double[][] expectedResults = new double[size][1];

        Random r = new Random(SEED);

        for(int i = 0; i < size;  i++){
            dataSet[i][0] = r.nextDouble();
            dataSet[i][1] = r.nextDouble();
            if(dataSet[i][1] < m * dataSet[i][0] + c)
                expectedResults[i][0] = POSITIVE;
            else expectedResults[i][0] = NEGATIVE;
        }

        double[][][] result = {dataSet, expectedResults};
        return result;
    }
}
```