UNIVERSITY of BRADFORD

Faculty of Engineering and Informatics
School of Electrical Engineering and Computer Science

Linear-separation based perceptron network construction

Final Year Project Report

Denis Torgunov

A thesis submitted in part fulfilment of the degree of BSc (Hons) in Intelligent Systems and Robotics

Supervisor: Dr. Attila Csenki

April 21, 2016

Declaration

The candidate confirms that the work submitted is his/her own and that appropriate credit has been given where reference has been made to the work of others. The candidate agrees that this report can be electronically checked for plagiarism.

Denis Torgunov

Contents

1 Abstract

This report discusses the design and implementation of a neural network training algorithm based on constructing a complex binary classifying neural network out of simple, linearly separable hyperplanes. It examines the other approaches that exist and cover the same problem domain and highlights how they differ from the algorithm utilised here. Finally, the resulting system is evaluated by examining its accuracy and ability to generalise on well-known open data sets.

2 Introduction and Background

This section describes what the project aims to achieve, and presents some background information on the algorithm being implemented. It also examines other approaches applicable to the same problem domain.

2.1 Project Description

This project is the first implementation of a system for binary classification of data that utilises a neural network constructed from unions and intersections of simple linearly separating hyperplanes, based on the work of Dr. Attila Csenki [1]. It is written in the Haskell programming language. Its performance is demonstrated and visualised for a 2-dimensional data set, and the accuracy measure under 10-fold cross-validation is given for some freely available data sets.

2.2 Aims and Objectives

Aim

To fully implement and evaluate the performance and error rate of a system for binary classification based on unions and intersection of simple linear separating hyperplanes

Objectives

- Implement a prototype for the system
- Evaluate the performance against existing approaches
- Refine the approach, adding pre-processing techniques and other optimisations
- Compare to other solutions to the classification problem
- Implement a way to visualise the algorithm's operation to make analysis easier

2.3 Background

This section considers the problem domain of binary classification, and presents a few existing approaches, which are based on machine learning, to said problem. It then considers the algorithm proposed by Dr. Attila Csenki and certain decisions based in background reading that influenced the final prototype produced.

2.3.1 Binary classification applications

The main goal of this project is to implement a binary classification system. In general, the classification problem entails finding a model or function, called a "classifier", that will assign a class or label to any given point [2, p. 29]. In the case when there are only two such classes, it is called a "binary classification problem" [2, p. 553]. For the purposes of the algorithm presented here, those sets (or "classes") are referred to as $+1.0$ and -1.0, but they can represent a variety of labels, such as a diagnosis ("positive" or "negative") or a flower species, to name a few examples.

Binary classification has an applications in a variety of fields, such as:

- medical diagnosis [3]
- quality control [4]
- evaluation of qualitative properties, such as demographic information [5]

There exist a variety of approaches utilising machine learning that are applicable in the same problem domain as the system described in this report. We shall examine some of those approaches in more detail in order to provide context to the problem, as well as to show how this approach is different from them.

2.3.2 Machine Learning

Machine learning is a multidisciplinary field that brings together computer science, mathematics and statistics [6], and can be viewed as a type of pattern recognition [7], now commonly used to "make sense" of large volumes of data by learning from the data itself [8].

One common and vastly applicable class of problems solved with machine learning deals with inferring, from a collection of input-output pairs, a function that maps inputs to outputs and can predict the output for new inputs [9, Ch. 18, p. 693]. The approach to binary classification implemented as part of this project falls into this category, with outputs being restricted to two possible "labels", designated $+1.0$ and -1.0. In the following sections, we consider some other approaches to solving this problem and compare them to the system produced as part of this project.

The approach to learning considered in this paper falls into a category known as "supervised learning", that assumes that the "training" data (i.e. the data used to "teach" the system) has correct labels assigned to it prior to training [9, Ch. 18, p. 695].

This means that the system learns by example, rather than trying to draw its own conclusions from the data presented to it.

2.3.3 Decision Trees

Decision trees reach a decision by performing a number of test on the attributes, with branches representing values of attributes and leaves representing the decisions returned by the tree [9, Ch. 18, p. 698].

Many manuals written for human use tend to resemble decision trees [9, Ch. 18, p. 698], which makes this mode of representation natural for us and easier to understand and interpret, and makes it possible for a human to understand the reason a given output was returned for a given input. Decisions trees, however, can be expensive when continuous input functions are involved [9, Ch. 18, p. 707].

The algorithm implemented by us focuses on separating points into classes using their position in n-dimensional space, effectively considering all of the attributes together as opposed to looking at them one-by-one, as is the case with decision trees. Moreover, the continuous nature of attributes is not an issue under this model.

2.3.4 Neural Networks

Conventional neural networks attempt to mimic the way the human brain processes information and recognises patterns [10]. It achieves this by creating artificial neurons, which attempt to replicate the way human neurons operate, and are trained by adjusting a weight value assigned to each individual input [11].

The beginning of neural networks can be traced to the McCulloch-Pitts perception, a simple binary classifier that assigns equal weights to all of its inputs [12]. The approach used by this system is based on an extension of that idea, developed by Rosenblatt and, later, Widrow and Hoff, which operates by assigning a value (a "weight") to each individual input, and producing a (binary) output, namely $+1$ or -1 [13, 14]. There also exists a learning rule for such perceptrons [13, 14], but we provide our own method of constructing one, described in section 2.4.1, along with a more detailed treatment of the way perceptrons operate.

Several different types of neural networks exist, differing in their approach to topology, utilising different activation functions and training rules [11]. The approach explored in this project generates topology as part of the training process, by combining several linear separators in various ways. The process is described in detail in section 2.5.

The way weights of a neural network are determined is based on mathematical algorithms, but the rationale behind why specifics inputs end up with specifics weights is often left unclear, as they are determined by training algorithms, making the network itself function as something of a "black box" [9, Ch. 18, pp. 727–737]. In the case of the algorithm proposed here, the reasoning behind assigning the weights is clear, as all

perceptrons serve to either construct a separating hyperplane between 2 points, or to combine several classifiers together.

Finally, we must note that neural networks are prone to overfitting (creating classifiers that generalise badly to inputs not encountered during training) in cases where the data that is being classified has too many parameters [9, Ch. 18 p. 736].

2.3.5 Support Vector Machines

Support vector machines also construct separating hyperplanes between classes, but in addition they aim to maximise the margin of separation [10, Ch. 6, p. 297]. This approach is then extended, for example by mapping the point into a higher-dimensional feature space to introduce extra flexibility [2, Ch. 21, p. 530], in order to deal with non-linearly separable patterns.

This approach is similar, at its core, to the approach used in this project. However, instead of always aiming to maximise the margin of separation, we leave the relative positioning of the separators determined via an extra variable, which can be adjusted during training to achieve, for example, the most optimal classification "gain" (i.e. classify as many misclassified points as possible), or reduction of a specific type of error. However, the value of this parameter is fixed at present, and its adjustment is subject to future development.

2.4 Building networks from separating hyperplanes

This section describes the basic mathematics behind constructing separating hyperplanes, and combining them, using unions and intersections. Using the techniques presented in this section, we can construct an arbitrarily complex network out of simple linear components.

This section is based mainly on [1], as well as discussion between the author of this paper and Dr. Csenki.

2.4.1 A linear separator

To begin with, we define a simple linear separator that can be used to classify two distinct points.

Given two points, expressed by a pair of vectors, $\vec{u}$ and $\vec{v}$, in $\mathbb{R}^n$, and a scalar value $c \in (0; 1)$, we can construct a hyperplane, H_c, based on Hesse normal form, such that [15]:

$$H_c(\vec{v}, \vec{u}) = \{x \in \mathbb{R}^n : \langle \vec{x}, \vec{v} - \vec{u} \rangle = c\|\vec{v} - \vec{u}\|^2 - \|\vec{u}\|^2 + \langle \vec{u}, \vec{v} \rangle\}$$

This hyperplane lines orthogonal to the line connecting the two points, and c is a scalar ($0 < c < 1$), determining the position of the plane relative to the $+1$ points. The smaller c is, the closer H_c is to $\vec{v}$.

Rearranging this, we arrive at

$$\langle \vec{x}, \vec{v} - \vec{u} \rangle - c\|\vec{v} - \vec{u}\|^2 + \|\vec{u}\|^2 - \langle \vec{u}, \vec{v} \rangle = 0 \tag{1}$$

Thus, for any $\vec{x}$ lying on the hyperplane H_c, the equation is equal to 0. However, if $\vec{x'}$ is "above" the plane, the equation will yield a positive value, and a negative value for a a vector "below" the plane. The direction of "above" and "below" are determined by the points $\vec{u}$ and $\vec{v}$, designated as representing a -1 point and a +1 point, respectively.

Using the sign(x) activation function, we thus obtain the appropriate classification for any point, assuming the two classes are $+1.0$ and -1.0.

2.4.2 Intersection and union of two classifiers

Given two classifiers, P_1 and P_2, we can arrive at a new classifier P_3 by taking an intersection or union of the two original classifiers. Note, that P_1 and P_2 need not be individual separating hyperplanes. They can also be themselves unions or intersections.

We will take $P_n(\vec{x})$ to mean "classification returned by the the classifier P_n".

Both of the definitions are based on [1].

Intersection

An intersection of two classifiers, P_1 and P_2, is such a classifier P_3 that:

$$P_3(\vec{x}) = +1 \iff (P_1(\vec{x}) = +1 \wedge P_2(\vec{x}) = +1)$$

We write this as

$$P_3 = P_1 \cap P_2$$

Assuming sign(x) as the activation function:

$$P_3 = \text{sign}\left(P_1(\vec{x}) + P_2(\vec{x}) - \frac{1}{2}\right)$$

Union

A union of two classifiers, P_1 and P_2, is such a classifiers P_3 that:

$$P_3(\vec{x}) = +1 \iff (P_1(\vec{x}) = +1 \vee P_2(\vec{x}) = +1)$$

We write this as

$$P_3 = P_1 \cup P_2$$

Again, assuming sign(x) as the activation function:

$$P_3 = \text{sign}\left(P_1(\vec{x}) + P_2(\vec{x}) + \frac{1}{2}\right)$$

We note here, that the intersection can be viewed as a kind of "union" for the -1 points. It classifies a point as -1 if and only if both P_1 and P_2 classify it as such.

2.4.3 Classifiers as perceptrons

The expressions above are sufficient to produce a working prototype of the system, and were, in fact, used by the first iterations of prototypes. However, in order to produce a perceptron network, the classifiers need to be converted to corresponding perceptrons, and the network itself needs to be constructed.

We begin by discussing what a perceptron is and the way it handles classification. According to [10, pp. 40-41], a perceptron (or neuron) can be viewed as a set of (synaptic) weights, denoted $w_1, w_2, w_3, \ldots, w_n$, a bias, denoted b, and an activation function φ. Applied to inputs $x_1, x_2, x_3, \ldots, x_n$, such a perceptron yields the output y. This can be achieved by taking the weighted sum u:

$$u = \sum_{i=i}^{n} w_i x_i$$

and calculating

$$y = \varphi(u + b)$$

This can be expressed using a diagram as shown in figure 1 (based on [10, Fig. 5, p. 41]):

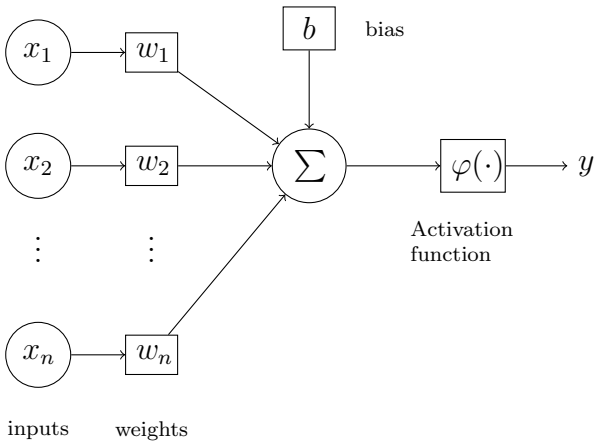


Figure 1: Perceptron diagram

Haykin mentions that the bias b can be viewed as applying an affine transformation to the weighted sum u and an activation function φ as the effect of limiting the amplitude of the output to a specific range of values [10, p.41].

For the purposes of binary classification as employed by this project, we assume the activation function to be the sign function, defined as:

$$\varphi(x) = \text{sign}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

Next, we show that each individual perceptron, assuming the activation function is known, can be expressed as a vector of weights, $\vec{w}$.

The output of the perceptron, y, is decided by the value passed to the activation function, $u + b$. We show that for $H_c(\vec{v}, \vec{y})$, $P_1 \cap P_2$ and $P_1 \cup P_2$ we can find such a vector $\vec{w}$ that, for any input vector $\vec{x}$, we can find $u + b$. Given the definition of u above:

$$u + b = \sum_{i=1}^{n} w_i x_i + b$$

let us introduce a new weight, $w_{n+1} = b$, and an extra input $x_{n+1} = 1$. Then:

$$u + b = \sum_{i=1}^{n} w_i x_i + 1 \cdot b = \sum_{i=1}^{n} w_i x_i + x_{n+1} w_{n+1} = \sum_{i=1}^{n+1} w_i x_i$$

If the numbers $w_1, \ldots, w_n, w_{n+1}$ are taken to form a vector $\vec{w}$ and $x_1, \ldots, x_n, x_{n+1}$ are taken to form the vector $\vec{x'}$, then, by the definition of vector dot product:

$$u + b = \sum_{i=1}^{n+1} w_i x_i = \sum_{i=1}^{n+1} x_i w_i = \vec{x'} \cdot \vec{w}$$

Thus we can characterise any perceptron, with the activation function $\text{sign}(x)$, by its weight-vector $\vec{w}$, with the last coordinate of the vector corresponding to the bias b, and the rest of the coordinates, forming a vector $\vec{w'}$, are taken to be the synaptic weights of the inputs. Then, to classify any given vector $\vec{x}$ by perceptron $\vec{w}$, we form an "augmented" vector $\vec{x'}$, by adding 1 as an extra coordinate to $\vec{x}$, in order to account for the unit bias. Then, the classification of $\vec{x}$ is given by:

$$y = \text{sign}(\vec{x'} \cdot \vec{w})$$

Now we can express each of our classifiers as perceptrons. From equation 1, knowing that the dot product in an inner product in Euclidean space [16, Ch. 7, p.540]:

$$\vec{w'} = \vec{v} - \vec{u}$$

$$b = -c\|\vec{v} - \vec{u}\|^2 + \|\vec{u}\|^2 - \langle \vec{u}, \vec{v} \rangle$$

The weight vectors for the union and intersection of two perceptrons are given in [1]. They are as follows:

$$\vec{w}_{\text{intersection}} = (1, 1, -1/2)$$
$$\vec{w}_{\text{union}} = (1, 1, +1/2)$$

Being able to generate individual perceptrons, we now concern ourselves with constructing the network itself. We note that a network can be expressed as a tree of vectors. The combinators (unions and intersections) will form the branched nodes, and the individual separators will be placed at the leaves.

Given a vector $\vec{x}$ to be classified, we add a single branch at each leaf (i.e. separating hyperplane), holding $\vec{x}$. Then, we "reduce" the tree by applying the following operation to the root until a classification is returned:

- If the tree is a single leaf, terminate. The classification is the coordinate of the one-dimensional vector at the leaf
- If the root of the tree has a single child, which is a leaf, take the vector at that child, $\vec{c}$, and replace the value at the root node with the dot product of $\vec{c}$ and the current value at the root node, $\vec{h}$: $(\vec{c} \cdot \vec{h})$. $\vec{c}$ is taken to be an augmented input vector, and $\vec{h}$ is the weight vector of a perceptron. Then, remove the child node. This represents a separating hyperplane
- If the root of the tree has multiple children, and all of them are leaves, construct a vector out of the values at the leaves, adding 1 as a unit bias. Then, replace the value at the root node with the dot product of this vector and the current value at the root node, removing the child nodes. This represents intersection and union perceptrons
- If the root has multiple children, and at least on of them is not a leaf, apply the whole process to each child

The perceptron network constructed this way yields the same classes as the "function" representation of classifiers described above. See section 6 for the testing approach used to verify this property.

2.5 Classifier construction

This section takes a look at the detailed algorithm that can be used to construct an arbitrarily complex classifier using linear separators and classifier combinations described above, as well as the perceptron networks constructed based on them. First, we describe the simple, easy to implement approach, and then discuss possible optimisations and refinements that can be applied to it and have been implemented as part of the prototype. A few other considerations and suggestions for improvement can be found in section 9. It should be noted that most of the thoughts and ideas in this section come from discussions with Dr. Attila Csenki [15].

Given a collection of points in n-dimensional space, each labelled as either +1 or -1, represented as $\vec{x_l} \in \mathbb{R}^n$ (where $l \in \{+1; -1\}$ is the label, or class), we can construct a perceptron network that classifies any given point in that space as +1 or -1, by the following sequence of steps:

1. Start with an empty network, $N =$ empty
2. Pick a pair of points belonging to different classes, $\vec{a}_{+1}$ and $\vec{b}_{-1}$
3. Construct a separator, $H_c(\vec{a}_{+1}, \vec{b}_{-1})$, to correctly classify those two points. We will call this classifier N_{minus}
4. Using the list of -1 points, filter out the points that are correctly classified by the resulting network. If no points remain, skip to 6
5. Pick a new point, $\vec{b'_{-1}}$, from the filtered list, and amend the classifier, so that $N_{\text{minus}} = H_c(\vec{a_{+1}}, \vec{b'_{-1}}) \cap N_{\text{minus}}$. Then return to 4
6. Take the network so far, and "unify" it with the network constructed during steps 3–5, so that $N = N \cup N_{\text{minus}}$
7. Remove the point $\vec{a}_{+1}$ from the list of points considered, and return to 2 unless there are no +1 points left, in which case we are finished

This is how the algorithm was first implemented for the very first prototype of the system. There are several notable shortcomings to this simplistic approach:

- A sub-network N_{minus} is constructed for every +1 point in the data set. This leads to networks that are much more complex than they need be. As part of optimising the algorithm, a few approaches have been considered to avoid this unnecessary complexity. But as the development process has shown, a naive approach (described in section 5.3) yields an erroneous network and leads to loss of accuracy. However, the issue has been dealt with later in the development process, yielding a more optimised network construction process that checks whether it now correctly classifies all of the training data every time a new classifier is added
- The means and criteria for choosing $\vec{a}_{+1}$ and $\vec{b}_{-1}$ are left unclear. There is potential for further customisation there, as different initial pairs will produce different networks. For the purposes of this report, the +1 points are considered "in sequence" (i.e. the order in which they appear in the input list), and the -1 points are chosen based on their Euclidean distance to the chosen +1 point
- The parameter c, which influences the position of the separator, is not defined. For the purposes of this project, it is chosen to be 0.5, however, some other approaches have been considered and are described in section 9

In addition, certain pre-processing of the data can take place. One example of such pre-processing is determining an initial separator, instead of starting with an empty network. This option has been implemented, as described in section 5.3, by using an initial separator $H_c(\overline{X}_{+1}, \overline{Y}_{-1})$, where $\overline{X}_{+1}$ is the centre of mass of all the $+1$ points in the data set, and $\overline{Y}_{-1}$ is the centre of mass of all the -1 points. Those are simply vectors, whose coordinates are the mean averages of the corresponding coordinates of all the $+1/-1$ points. Such a separator can have both a positive and a negative impact on accuracy, based on the data set, as can be seen in section 8. A schematic representation of the whole process, including the validation step discussed in the next section, is show in figure 2.

2.6 Classifier validation

One of the aims of this project is to evaluate the performance of the classification neural networks produced by the algorithm in question, and to enable future users of the system to test its accuracy on various data sets. To that end, the GUI will provide feedback to the user by presenting them with a confusion matrix, as described in [17], listing the amount of true and false positives/negatives, as classified by the network, as well as an accuracy measure, which is computed as

$$\text{Accuracy} = \frac{\text{Number of true positives} + \text{Number of true negatives}}{\text{Total number of data points}}$$

The terminology "true/false positive/negative" is used in this report, to keep it consistent with the commonly accepted terminology [17]. In the context of the binary classifiers implemented here, a "positive" is any $+1$ point, and a "negative" is any -1 point.

In order to assess the error and accuracy of a model, and to evaluate how well it generalises to data not seen during training, the data available is usually split into at least 2 subsets [8, Ch. 7, pp. 219–260], called "training" and "validation". The idea behind this process is to evaluate how the classifier will behave when presented with new data not encountered during training (generalisation).

The question then becomes how to appropriately split the data into the two sets. Two approaches have been considered by the author of this paper, and both are available in the final prototype. Those approaches are 70/30 hold-out validation and 10-fold cross-validation.

In both approaches, we begin by shuffling the data, changing the positions of data points in the list at random.

hold-out validation

The simplest form of validation, hold-out validation splits the shuffled list into two parts: $p\%$ of the list are assigned to training, and the remaining $(100 - p)\%$ are left for validation. The system produced uses 70% of the data set for training, when using hold-out validation. Although this number could easily be changed

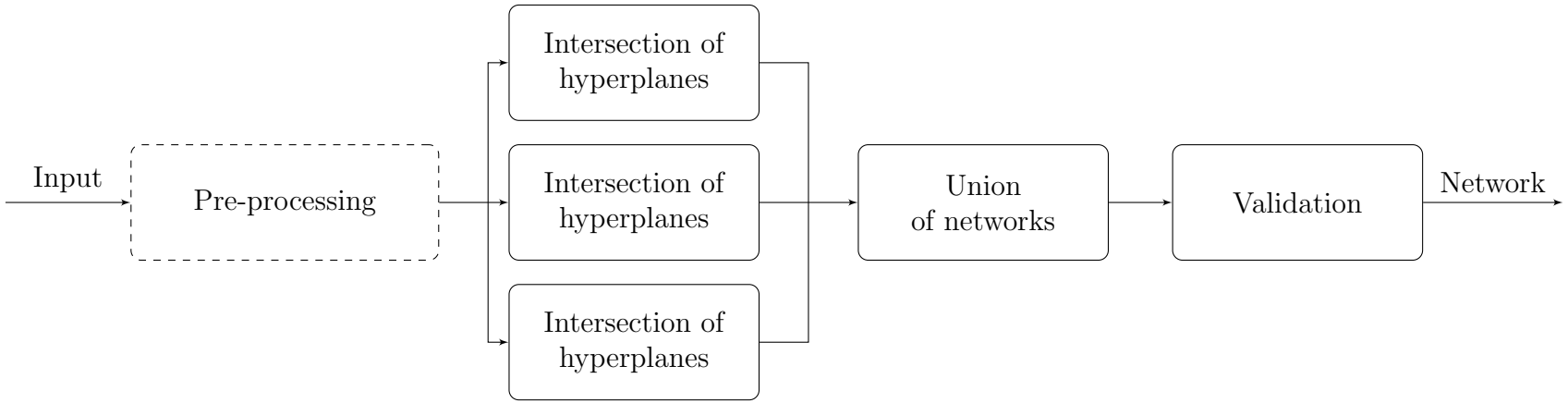


Figure 2: Basic approach to constructing a classifier

or even set in the GUI, the decision has been made to "hard-code" it at present, to make the UI simpler to test and use, limiting user input to choosing from available options.

k-fold cross-validation

Unlike hold-out validation, cross-validation attempts to utilise as much of the data set as possible during both training and validation, since the amount of data available might be sparse [8, Ch. 7, pp. 219–260]. To achieve this, the shuffled data list is split into k equal-sized parts. One of those k sub-lists is then "set aside" for validation, and the system is trained using the remaining sublists. The process is repeated k times, and the results are averaged before being presented to the user. In the event that the size of the data is not evenly divisible by k, we do our best to approximate by allowing the first ($l \mod k$) (where l is the size of the data set) sublists to have one element more than the remaining sublists. Much like with hold-out validation, although k can be any number up to the size of the data set, it is kept at 10 for the purposes of the prototype.

3 Project Management and Tools

This section considers the appropriate choice of development methodology for the project. It also details the tools and libraries that have been chosen prior to starting the project (or a specific prototype iteration) to help achieve the objectives presented.

3.1 Methodology

Rapid development with evolutionary prototyping has been chosen for this project. The nature of the project lends itself well to this style of development due to its research flavour. Rapid prototyping has been shown to have positive effect on the process of software development [18]. The main problems arising from picking this approach seem to be poor design and maintainability, but those can be mitigated by clearly defining the purpose and scope of each prototype, as well as keeping the software modular. The modular design of this project is further discussed in section 4.

Rigid models, such as Waterfall, are not feasible to be used under the conditions imposed by this project. They tend to favour formality and strict adherence to the development "cycle" [19] and make the it harder to dynamically change requirements [20], whereas this project focuses much more heavily on exploring, evaluating and adapting the system being developed. Moreover, waterfall assumes the full requirements of the system are known in advance [21], which is not true for this project.

The various agile methodologies have many similarities to our chosen approach, but seem to be at their most effective when used in a team setting, as opposed to a lone developer [22]. Most of the benefits of agile methodologies that focus on quick

results and client communication are shared by RAD (rapid application development) methodologies, but agile seems to focus more on making teams more efficient.

Over the course of the project's development successive prototypes have been demonstrated to and discussed with the project's academic supervisor. Those discussions have influenced the development of the project and helped shape the final feature set of the project. In addition, some other possible improvements have been discussed but not implemented at this stage. They are discussed in more detail in section 9. Section 5 discusses various prototypes produced as part of developing this project, focusing on the new features added at each stage of development.

3.2 Tools and Libraries

3.2.1 Haskell Implementation and Compiler

There exist several implementations of the Haskell programming language [23]. For the purposes of this project the Glasgow Haskell Compiler (GHC) has been chosen. It is the "de facto standard" [23] Haskell compiler, and has a wider selection of libraries [24]. Including those that can be used for plotting graphs [25] and adding a Graphical User Interface (GUI).

In addition, the Glasgow Haskell Compiler supports code optimisation [24], which could become a factor if the system's time efficiency is evaluated in addition to the results it produces. The HaskellWiki also suggests that, while the interactive environment in Hugs, a commonly used Haskell interpreter, runs quicker, the code produced by GHC runs a lot faster, as a result of the optimisations carried out [23]. The presence of an interactive mode as well as compilation mode in GHC makes it possible to debug the code written by manually running newly implemented functions, and facilitates experimentation during development, allowing the programmer to prototype the functions interactively, observing the behaviour of various pieces of code before making specific decisions.

3.2.2 Parsec

In order to make it possible for the system to use a wide variety of data sets, we use the Haskell Parsec library to implement simple parsing capabilities. It is a monadic parser combinator library that enables us to write simple and flexible parsing systems [26].

This project implements parsing for CSV (comma-separated values) files, one of the common data formats for open data sets. Other similar formats can be converted to CSV by third-party tools.

The parser used in this project is based on the insightful chapter on parsing in the book "Real World Haskell" [27, Ch. 16].

3.2.3 GTK+ and Gtk2hs

In order to make it easier to use this system, a Graphical User Interface has been added to the Haskell library produced. In order to make the system cross-platform, the GTK+ GUI toolkit has been chosen as the GUI system of choice. It is written in C and available for Linux, Windows and Mac OSX [28].

In order to use the GTK+ libraries from a Haskell program, we make use of Gtk2hs, the Haskell GTK+ bindings [29]. Gtk2hs is known to work well with Linux and Windows, however the documentation relating to compiling and using it under Mac OS X is outdated at the time of this writing [30].

In addition, a tool called Glade [31] has been used to construct the layout of the GUI. It allows the user to use Drag-and-Drop functionality to produce an XML file, which can then be loaded by any program utilising GTK+ and used to represent any window layout. This allows us to keep the positioning and functionality of GUI elements separate, as long as we stick to consistent naming.

3.2.4 Chart

One of the objectives laid out at the start of this project was to implement some form of visualisation of the network's behaviour. The final prototype produced possesses that functionality, allowing the user to (optionally) display a simple plot of the data, as well as a sample of points in space classified by the network in specific ways, which allows the user to broadly see what areas are classified as $+1/-1$.

In order to make this behaviour possible, the Chart library has been used [32]. It allows for quick and easy construction of various kinds of graphs, and provides a GTK+ back end [33] that makes it easy to use it with Gtk2hs in order to display the plots to the user.

3.2.5 QuickCheck

For testing, we use Haskell's QuickCheck tool, which allows us to specify properties of functions and then test these properties on automatically generated test cases [34]. This approach can be combined with formal verification for critical sections of the program [35, 36] to ensure robustness and stability. At the same time, it supplements formal verification in order to allow the programmer to focus on formally proving only the most critical properties of a system, and allowing QuickCheck to handle the rest.

QuickCheck facilitates testing by allowing the programmer to specify a collection of properties the system is assumed to have if it is bug-free [34]. Those properties can be similar or identical to the ones that can be used to formally prove program correctness, as long as they are decidable [36]. QuickCheck can then generate random values for the function arguments (of the correct type, by leveraging Haskell's type system) and attempts to falsify the property [34]. Assuming enough random values have been used for testing, it gives the programmer good assurance that the property holds.

However, such an approach can still fail to generate a valid counterexample to the property, even if one exists, if such a counterexample is too rare [36]. For this reason, it is recommended to use formal verification grounded in mathematics to prove the properties of the program that are crucial to its execution.

The way this project leverages QuickCheck for testing is described in more detail in section 6.

3.2.6 Cabal

Cabal is a system used to configure and build Haskell applications and libraries [37], similar to the way Makefiles and automake tools can be used for configuring and building C and C++ applications.

This project uses Cabal to make it easy to build the system and ensure all the dependencies required for compilation are present. For details on how Cabal can be used to assist in the deployment of the system, see section 7.

However, it should be noted that Cabal should only be used for building software, and not installing it on the user's machine, although it does have the functionality that allows it to do so, using a related tool known as cabal-install [37]. Instead, the operating system's preferred way of managing software installations should be used [38].

3.2.7 Haddock

In addition to this document, which outlines the creation and implementation of the project, the source code itself is annotated to make maintenance and future development easier. The comments adhere to Haddock syntax, which allows them to be exported to HTML documentation [39], which is bundled with the submitted source code.

4 Design

During the early discussions with the academic supervisor regarding the structure and deliverables associated with this project, it has been decided that a big focus in the development process would be identification and addition of possible improvements to the general algorithm. In order to facilitate the development of such a system, as well as to make it possible to identify how changes made to the algorithm's operation affect the outcome (i.e. the generated model), it was necessary to make the structure of the project modular, allowing the library users to "cherry pick" which optimisations were to be used during network construction. Different modules represent various elements that can be changed and adjusted.

Moreover, to make it easier to carry out optimisations on the original implementation of the algorithm, originally training algorithms have formed a set of modules, with each module corresponding to a particular implementation version. That made it easier to spot if any of the optimisations carried out result in a decrease in accuracy or

any other unexpected changes to the results. This approach has paid off, allowing the developer to locate bugs that led to decreases in accuracy throughout development. The final version has dispensed with versioned training algorithms, as the current existing implementation appears to be the most optimal out of those considered. As such, there are modules, `Training.Version1` and `Training.Version2`, that used to be present in early iterations, but have been removed from the final version.

The following modules form the main part of the algorithm:

- `Types`: This module holds type synonyms that are used throughout the project, to make it easy to import and use them. The use of those type synonyms makes type signatures cleaner and easier to read.
- `Networks`: This module implements the `Network` data type, which represents a neural network. The details of how the network is constructed are not exported by this module. Instead, the functions to construct specific types of networks are exported, such as union and intersection networks [1], as well as individual separating hyperplanes.
- `InitialSeparators`: This module implements "initial separation" functionality, wherein a simple separating hyperplane is constructed before the rest of the network is built, which aims to provide a good starting point with the majority of the training inputs already correctly classified. Examples include separation of centroids of the +1/-1 points.
- `Training`: The modules that perform the actual construction of the network. This used to hold a collection of different modules, such as `Training.Version1` and `Training.Version2`. While those modules are no longer present in the final version, they are mentioned for completeness, as they are referenced in some of the older prototypes.

The modules outlined above are used in network construction. The way those modules are interconnected can be seen on the diagram in figure 3:

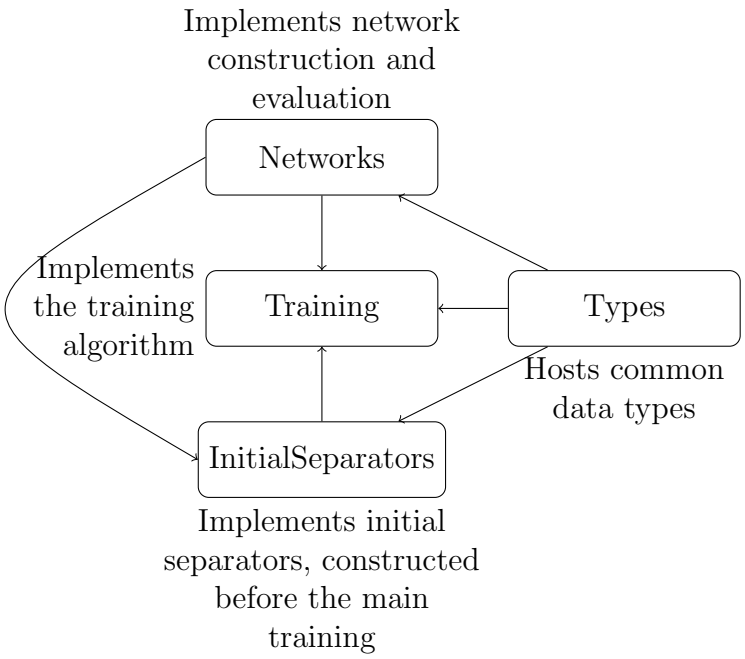


Figure 3: Modules used in network construction

The arrows on the diagram indicate which modules are imported by other modules.

The following modules are not directly involved in the network construction process, but instead serve to provide supporting functionality, such as model validation and reporting the results to the user:

- `Validation`: This module implements hold-out and k-fold cross-validation techniques in order to evaluate how the network performs on both training data, and data that it has not been exposed to during construction.
- `ConfusionMatrix`: This module is used to provide a concise summary of network performance, and is used by the `Validation` module when summarising results.
- `Parsing`: This module includes CSV parsing functionality, to help supply training and validation data to the program.
- `Gui`: This module uses the Gtk2hs Haskell GTK+ bindings in order to provide a simple GUI. It allows the user to tweak individual parameters (such as validation methods or initial separators) and presents a confusion matrix with results to the user, as well as plotting the visualisation of the network.
- `Plotting`: This module constructs the plots of the data and the network. In the event that the data has dimensionality higher than 2, it constructs several projections to display.

5 Implementation

As described in section 3.1, the implementation of the system took the form of several prototypes, each building on the previous one, and adding a small, manageable set of new features. This section describes various prototypes produced as part of the development process, as well as the additions introduced in each prototype.

The focus of this section is on the crucial decisions and optimisations made, as well as challenges faced during implementation. For full listing of the project's source code, please see the code submission.

5.1 Prototype 1

As outlined above, the `Types` module holds a collection of type synonym definitions that are used throughout the code base. This makes particularly long type signatures easier to read, and helps clarify what parameters or values are returned by functions represent.

For example, let us examine the `Weights` and `Input` types (the majority of the comments have been omitted for brevity):

```
type Weights = [Double]
type Input = [Double]
```

Both refer to the same type: a list of `Double`s. However, one represents an input vector (i.e. a point to be classified), while the other refers to the weights within a perceptron. Using a different name for the same basic type in different circumstances makes the type signatures more readable and self-contained.

Other crucial types have to do with training data:

```
type Classification = Double
type TrainingInput = (Input, Classification)
type TrainingSet = [TrainingInput]
```

A `TrainingInput` represents an input vector with a known classification, that can be used for the purposes of supervised learning. A training set is simply a collection of such inputs. Please note that the type signatures do not differentiate between "training" and "validation" sets, as the applications for both in terms of supported operations are virtually identical.

This early version of the system also utilised a concept of "paired inputs". Before the network is constructed, all $+1/-1$ points were formed into pairs using a specific "pairing function". It was hoped that different takes on choosing pairs of points to be compared could be attempted, but it was never implemented within the scope of the current final prototype. Thus, those concepts are not present in later versions of the training algorithm implementation.

The related types are as follows[1]:

[1] Please note that a red arrow indicates a linebreak inserted for formatting purposes

```
-- | 'PairedInputs' represent pairs of +1 and -1 points that will be used
   ↪ during network construction to construct separating
-- hyperplanes. It is a list of lists, with each list being a list of -1
   ↪ points paired to a specific +1 point.
-- Used in the early algorithm versions.
type PairedInputs = [[(TrainingInput, TrainingInput)]]

-- | A function that determined the order in which the points will be
   ↪ compared.
type PairFunction = [TrainingInput] → [TrainingInput] → PairedInputs
```

For the purposes of this prototype, the main way inputs are paired is based on the Euclidean distance, with the ones close together coming first in the list, as discussed earlier.

The main focus of the initial prototype was implementing the algorithm in the form it is described in section 2.5. However, it was also considered to be of potential use to have the ability to trace back the construction process, in order to see what hyperplanes were generated and in what order. To this end, a `Network` type has been introduced, that shows the topology of a generated network. This network could then be "run" on a set of data, using the definitions presented in section 2.4. This allowed for the testing of the network's performance before the conversion to perceptrons was introduced.

A `Network` is defined as follows:

```
data NetworkDesc
    = Empty
    | Hyperplane { plusPoint :: Input
                 , minusPoint :: Input
                 , c :: Double
                 }
    | Union NetworkDesc NetworkDesc
    | Intersection NetworkDesc NetworkDesc
      deriving (Show, Eq)

type NetworkFunction = Input → Classification

data Network = Network { f :: NetworkFunction
                       , net :: NetworkDesc
                       }
```

The `NetworkDesc` datatype fulfils the function of "keeping track" of network's construction, outlined above. Since at every recursive step, either a union or an intersection of networks is produced, by removing `Union` and `Intersection` data constructors we can see the succession in which the separators are generated, and how they are combined.

A `NetworkFunction` mimics the action of a perceptron network on an input vector. A later prototype introduces a function to convert a `Network` to a `PeceptronNetwork`, but the early prototypes simply "simulated" the perceptrons. For example, "running" a `(Hyperplane p m c)` is equivalent to evaluating the expression

$$\text{sign}(H_c(\text{p}, \text{m}))$$

and "running" a `(Union p1 p2)` amounts to evaluating

$$\text{sign}\left(\text{run}(\text{p1}) + \text{run}(\text{p2}) + \frac{1}{2}\right)$$

where $\text{run}(p)$ is the result of classifying the input data with classifier p.

We also define a function `runNetwork`, that lets us classify any given `Input` with a given `Network`.

```
runNetwork :: Network → Input → Classification
runNetwork n xs = (f n) xs
```

With those functions present, it is possible to implement the simple network construction algorithm.

```
seave :: Network → (TrainingInput, [TrainingInput]) → (TrainingInput, [
    ↪ TrainingInput])
seave net (x,ys) = (x, filter misclassified ys)
    where
      misclassified :: TrainingInput → Bool
      misclassified (point, c) = ((runNetwork net) point) /= c

augmentNetwork :: Network → (TrainingInput, [TrainingInput]) → Network
augmentNetwork net (_, []) = net
augmentNetwork net (x, (y:ys)) = augmentNetwork newNet (seave newNet (x, ys)
    ↪ )
    where
      newNet = net `intersectNet` (hyperplane (fst x) (fst y) 0.5) -- c =
          ↪ 0.5 hardcoded

createPlusNet :: [(TrainingInput, TrainingInput)] → Network
createPlusNet tis = augmentNetwork emptyNet (seave net simplifiedTis)
    where
      simplifiedTis = simplify tis

unifyNetwork :: [Network] → Network
unifyNetwork (net:nets) = foldr unionNet net nets

createNetwork :: TrainingSet → Network
createNetwork ts = unifyNetwork $ map createPlusNet preparedInputs
```

```
  where
    preparedInputs = pairInputs distanceBased ts
```

Referring to the algorithm outlined in section 2.5, the function `augmentNetwork` is used to generate the N_{minus} for each +1 point, by adding an intersection of the network generated so far and the network separating a pair of misclassified points. Point `x` represents the +1 point under consideration, and `y` represents the closest misclassified −1 point. `ys` refers to the list of other currently misclassified −1 points. Where the list of `(y:ys)` is empty, the recursion terminates. Otherwise, the `seave` function is used, to remove the points which are now correctly classified from the list of points considered.

`createPlusNet` is used as the starting point for the recursive `augmentNetwork` function. It is mapped over the list of all points, yielding the intersection network based around every +1 point, which are then joined via the union combinator using the `unifyNetwork` function.

The entry point for this process is the `createNetwork` function, the only function exported by this module.

At this stage, very simple validation has been implemented, using 70% hold-out validation, and printing to the terminal the count of misclassified points from the training set.

5.2 Prototype 2

Parsing

This version of the prototype introduced parsing of CSV data to be used for classification, allowing us to test on a wide variety of available data sets and to avoid having to supply training values in a separate module to be loaded, which is what was being done before. The parsing code makes use of the Parsec library and is based heavily on [27, Ch. 16]. The parsing code remains almost unchanged from this iteration onwards, and can be found in the `Parsing` module of the source code provided with this submission.

We must give our attention to the following function's type signature:

```
readCSVData :: FilePath → IO (Either String (ClassMap, TrainingSet))
```

This function is the primary function used to parse a file. Given a path to a CSV file, we return an `Either` value, wrapped in an `IO` monad. The monadic nature of this function has to do with the fact that the parsed data comes from a file. Let us examine the `Either` type in mode detail however.

`Either` is a Haskell data type that returns one of two possible types. In this case, either a `String`, signalling an error during parsing, or a `(ClassMap, TrainingSet)` pair.

In the event that the returned value is a `String`, we print it to the user in order to notify them that an error occurred trying to parse the data they supplied.

Otherwise, the pair `(ClassMap, TrainingSet)` is returned. The `TrainingSet` can be readily used to train and validate a network. The `ClassMap` is used to map whatever classes the data has by default (treating them as `String`s) to either $+1$ or -1 that are included in the data set. The inclusion of `ClassMap` allows us to notify the user of which class is represented by $+1$ and which by -1. It's definition, provided in `Types`, is as follows:

```
type ClassMap = [(String, Double)]
```

If there are not 2 distinct classes in the data supplied, we signal an error by returning a `String` (the `Left` value of `Either`).

Validation

This version of the prototype also introduced the module `Validation`, which implements both hold-out and k-fold cross-validation, and the module `ConfusionMatrix`, which is used to represent confusion matrices that are produced by the validation functions. The `ConfusionMatrix` module is not detailed in this report, as it is simple and straightforward, with sufficient documentation provided as part of the source code submission.

Firstly, we mentioned in section 2.6 that the lists need to be shuffled. This is achieved by employing the following functions:

```
shuffleList' :: (RandomGen g, Eq a) ⇒ g → [a] → [a] → [a]
shuffleList' _ [] accum = accum
shuffleList' g ss accum = shuffleList' g' ss' (s:accum)
    where
      (index, g') = randomR (0, (length ss - 1)) g
      s = ss !! index
      ss' = (take index ss) ++ (drop (index+1) ss)

shuffleList :: (RandomGen g, Eq a) ⇒ g → [a] → [a]
shuffleList g l = shuffleList' g l []
```

We use accumulator-based recursion in order to construct a shuffled version of the list. Note that we take a random seed `g` as a parameter. This seed is either generated randomly inside an `IO` monad (the approach used by final prototype), or supplied by the programmer for testing purposes. In the latter case, the order of the elements in the shuffled list will not, in fact, be truly random, but will be determined by the seed. This preserves Haskell's referential transparency property and keeps our code base purely functional and free of side effects whenever possible.

When shuffling, we use the built-in `randomR` function to generate a random number that could serve as a valid index into the list, as well as generating a new random seed. We then add the element at the given index to the head of the accumulator, and construct the list `ss'`, removing the element at the chosen index. We recursively perform this operation until the list of values is empty. The end result is a list containing

all the same elements as the list supplied, but in a different order. Randomisation of the order can be achieved by supplying a randomly generated seed `g`.

Next, we examine the `ValidationFunction` type:

```
type ValidationFunction = StdGen → TrainingSet → (TrainingSet → Network)
    ↪ → (ConfusionMatrix, ConfusionMatrix, Network)
```

This represents a function that can be used to classify a network. Its first argument is a random seed, used by the shuffling function above to randomise the order in which the points are considered. The second argument is the whole training set, and the third (and final) argument is a function used to train the network. The exact nature of this function has changed throughout development. In the early iterations it simply represented the `createNetwork` function discussed in the previous section, sharing its type signature. But the fact that it is taken as an argument by the validation function allows us to pass different versions of this function (from different prototypes) in, as well as add different initial separators to the construction process, without affecting the validation.

Finally, the validation function returns a tuple of three elements: a training confusion matrix, a validation confusion matrix, and a network. In the case of hold-out validation, the network trained on the p% of data is returned. In the case of k-fold cross-validation, a network trained on the entirety of the data set is returned.

The details of how hold-out and k-fold cross-validation have been implemented can be found in the source code submitted with this project.

5.3 Prototype 3

This version of the prototype introduced initial separator functionality. Moreover, as support for such separators needed to be added to the program, we have decided to take this opportunity to optimise the network construction by attempting to stop the algorithm early and avoid constructing a separate sub-network for each +1 point, as discussed in section 2.5.

However, the approach taken here did not involve enough forward thinking, and as a result a buggy iteration was produced. The bugs were eventually fixed in a later iteration, but we look at their causes here in order to highlight the fact that, using iterative prototyping, absence of clear planning for changes introduced can lead to unforeseen complications.

Initial separators

Firstly, we introduce a new type synonym to represent the function that constructs the initial separator:

```
type SeparatorFunction = [TrainingInput] -- ^ All of the +1 points
                       → [TrainingInput] -- ^ All of the -1 points
                       → Network
```

This function creates a `Network` (representing an initial separator), based on the two lists of inputs given. It is expected that the separator produced uses some form of metric gained from the points as a whole. The simplest possible separator we can introduce is a "dummy" separator, mimicing an absence of initial separation by simply returning an empty network, irrespective of inputs.

```
noSeparator :: SeparatorFunction
noSeparator _ _ = emptyNet
```

More interestingly, we also provide a separator that is based on the centres of gravity of points. The centre of gravity of a set of points is taken to be a point whose coordinates are the averages of the corresponding coordinates of the points in the set.

```
sumsOfCoordinates :: [Input] → Input
sumsOfCoordinates = map sum ∘ transpose

centroid :: [Input] → Input
centroid ps = map (/ numOfSamples) (sumsOfCoordinates ps)
    where
      numOfSamples = fromIntegral $ length ps

centroidSeparator :: SeparatorFunction
centroidSeparator plusOnes minusOnes = hyperplane (centroid plusOnes') (
    ↪ centroid minusOnes') 0.5
    where
      plusOnes' = map fst plusOnes
      minusOnes' = map fst minusOnes
```

The `sumsOfCoordinates` function produces a list whose `n`th element is the sum of elements at index `n` across the list `[Input]` provided. It is used by the `centroid` function to find a centre of gravity of a set of points. We can then use the `hyperplane` function (from the `Network` module, which simply produces a `Network` consisting of a single separating hyperplane) to construct the separator between the two centroids.

Optimisation attempt

This section focuses on the changes made to the original version of the training algorithm, discussed in section 5.1.

```
1 createPlusNet :: (TrainingInput, [TrainingInput]) → Network
2 createPlusNet tis = augmentNetwork net (seave net tis)
3     where
4       net = hyperplane (fst $ fst tis) (fst $ head $ snd tis) 0.5
5
6 augmentUnify :: Network → (TrainingInput, [TrainingInput]) → Network
7 augmentUnify net (_, []) = net
8 augmentUnify net is = net `unionNet` (createPlusNet is)
```

```
9
10 createNetwork' :: PairedInputs → Network → Network
11 createNetwork' [] n = n
12 createNetwork' (t:ts) n
13     | isEmptyNet n = createNetwork' ts (createPlusNet (simplify t))
14     | otherwise = createNetwork' ts (augmentUnify n misclassifiedInputs)
15     where
16       misclassifiedInputs = seave n (simplify t)
17
18 createNetwork :: SeparatorFunction → TrainingSet → Network
19 createNetwork sf ts = createNetwork' preparedInputs (sf plusOnes minusOnes)
20     where
21       preparedInputs = pairInputs distanceBased ts
22       plusOnes = filter (λ(x,c) → c == 1) ts
23       minusOnes = filter (λ(x,c) → c == (-1)) ts
```

The first change we make is to amend the type signature of `createNetwork` (line 18) to allow for an extra parameter: `SeparatorFunction`. The first bug arising from this implementation has to do with the use of this separator. Note that line 13 checks for the base case in the event that no initial separator is supplied, constructing the initial network much in the same way as the original version. Line 14, however, makes sure that if a network has been constructed (either through the presence of an initial separator or recursion), we first seave the data points with respect to that network (line 16), before proceeding.

The mistake made in this implementation was not keeping track of the initial configuration of points. This lead to the unfortunate situation where a point would be correctly classified by a network, sieved out, and then become misclassified by a point added in one of the subsequent iterations. But since the point is no longer present in the list, we do not notice the decrease in accuracy and thus do not amend the network as necessary.

This bug quickly came to light during testing, and was fixed in a subsequent iteration, as detailed in section 5.6. Since the interweaving prototype focused on constructing the GUI, and the bug didn't manifest itself for simple test cases, it was not uncovered straight away.

5.4 Prototype 4

Since the project now had many building blocks that could be mixed and matched (choosing validation methods, initial separators and versions of the training algorithms), and the introduction of basic plotting capabilities was planned, it was decided that a simple graphical user interface would need to be produced.

The interface needed to possess the following functionality:

- Allow the user to choose a data file to load

- Allow the user to change the parameters (algorithm version, initial separator, validation method) before network generation
- Allow the user to easily re-generate the network without having to navigate the file system in order to find the file again, if any of the parameters were changed
- Present the user with the confusion matrices for both training and validation data
- Display error messages and other communication to the user

In addition, provisions needed to be made to accommodate future ability to plot the networks.

The resulting GUI is shown in figure 4. A discussion of key challenges faced during development follows.

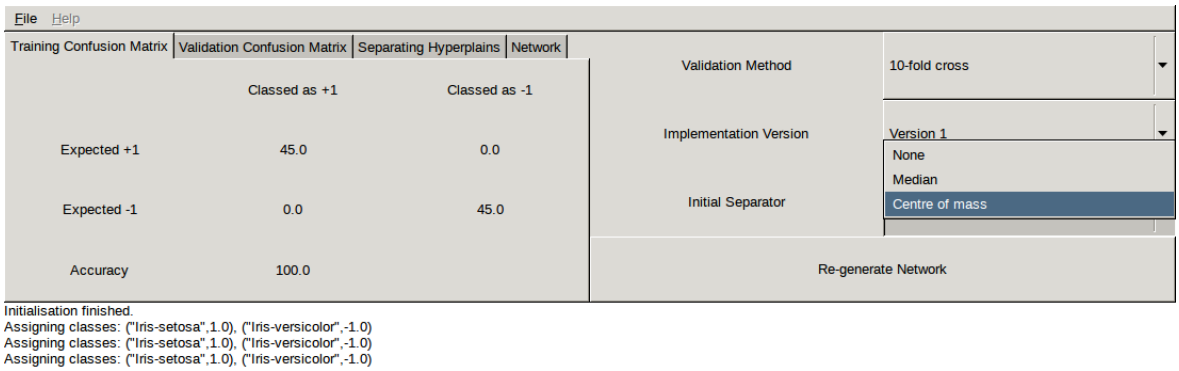


Figure 4: A screenshot of the initial version of the GUI

In order to make it easy to adapt the GUI as new elements are added, we have chosen to use Glade (as described in section 3.2.3) to generate the layouts. This produces an XML file with a `.glade` extension that can be read by the program. We made sure to keep the names of individual UI elements consistent. This paid off later, during the construction of the final prototype, as it allowed us to move the layout elements around without affecting functionality.

The main challenge in integrating GUI into a Haskell problem is the inherently impure nature of user's interaction with the UI. In order to keep it manageable, the `Gui` module was introduced, hiding the complexity of the UI's operation.

First, we needed to identify the components that the user needed to be able to change. Those have been abstracted into `Maps`, with the keys of the maps corresponding to the indexes returned by combo boxes for each choice:

```
separatorMap :: Map Int SeparatorFunction
separatorMap = fromList [ (0, noSeparator)
                        , (2, centroidSeparator)
                        ]

algorithmMap :: Map Int (SeparatorFunction → TrainingSet → Network)
algorithmMap = fromList [ (0, Training1.createNetwork)
                        , (1, Training2.createNetwork)
                        , (2, Training3.createNetwork)
                        ]

validationMap :: Map Int ValidationFunction
validationMap = fromList [ (0, crossValidation 10)
                         , (1, splitValidation 70)]
```

The main part of this module is a `prepareGui` function, which initialises the main window and sets up all of the control elements, such us buttons and menu items. Since different parts of the module needed access to different GUI elements in order to function, a single data type `GUI`, has been implemented to hold all the elements. A parameter of this type could be passed around throughout the module, with individual functions using its accessor functions to extract the elements they needed.

```
data GUI = GUI { consoleOut :: (String → IO())
               , displayTrainingMatrix :: (ConfusionMatrix → IO ())
               , displayValidationMatrix :: (ConfusionMatrix → IO ())
               , displayNetwork :: (Network → IO ())
               , dataFile :: IORef FilePath
               , algorithmVersion :: ComboBox
               , initialSeparator :: ComboBox
               , validationMethod :: ComboBox
               , rootWindow :: Window
               }
```

For example, a `dataFile` is an `IORef`, meaning it represents a conventional variable, whose value might be changed throughout process execution. When the user chooses to load a data file, the path to that file is stored in this variable. Whenever the user chooses to re-generate the network, the data is loaded from the location this variable points to. The `IORef` itself, which serves, effectively, as the address of the variable, is kept inside the `GUI` parameter, extracted by either the function handling menus, or the button callback, to set and read it respectively.

The other functions and elements are used in much the same way. Please consult the source code submission for details.

We note, as discussed earlier in section 2.6, that while it would be possible to allow the user to select the p% for hold-out validation, and the k for k-fold cross-validation, it would introduce additional complexity with validating user input. For this reason it has been decided to fix $p = 70$ and $k = 10$ at present.

5.5 Prototype 5

Up until now, the prototypes have only simulated the presence of a neural network, instead focusing on the details of how the network is trained. This iteration introduces the construction of actual perceptrons and networks (represented as trees), as described in section 2.4.3. The functions generating perceptrons are quite straightforward, relying on the definitions of $\vec{w}$ given in section 2.4.3. Instead, allow us to turn our attention to the `reduceLeaves` function, which implements the tree reduction method described:

```
1  type Weights = [Double]
2  type PerceptronNetwork = Tree Weights
3
4  leaf :: Tree a → Bool
5  leaf tree = null $ subForest tree
6
7  reduceLeaves :: PerceptronNetwork → PerceptronNetwork
8  reduceLeaves tree
9      | leaf tree = tree
10     | (length $ subForest tree) == 1
11       = Node { rootLabel = [sign $ (rootLabel tree) <.> (rootLabel $ head $
             ↪ subForest tree)]
12                , subForest = []
13         }
14     | and $ map leaf $ subForest tree
15         = Node { rootLabel = [sign $ (rootLabel tree) <.> ((concat $ map
               ↪ rootLabel $ subForest tree) ++ [1.0])]
16                , subForest = []
17                }
18     | otherwise = Node { rootLabel = rootLabel tree
19                        , subForest = map reduceLeaves $ subForest tree
20                        }
```

As you can see, this function is recursive. The recursive step (lines 18-20) simply retain the contents of the root node and map the function over the leaves of the tree.

We consider 3 base cases:

- Line 9: A tree that is a single leaf reduces to itself. This case arises whenever, for example, we have a union of a single hyperplane and an intersection. In this case, after one reduction, one of the branches (previously leading to a hyperplane and an input) will contain a leaf, while another (an intersection) will contain a sub-tree

- Lines 10-13: A tree with a single child node. This represents a hyperplane with an input as its child. The input is assumed to already be "biased", and so we simply prune it (by returning a node with no children) and set the value at the node to the result of applying the input to a perceptron at this node
- Lines 14-17: A tree with multiple children, all of which are leaves. This case represents a union or intersection node the children of which have all been reduced already. We collect the values at the child nodes into a list and "bias" it by adding a `1.0` to the end. Then, we apply that to our perceptron and prune the tree

This function avoids assuming that the union and intersection are binary operations, although they are at the moment. This has been done in order to make it possible to extend those combinators, allowing them to handle more inputs at once in the future, which should greatly reduce the complexity of the network. This is discussed further in section 9.

5.6 Prototype 6

After the GUI has been developed, it became apparent that the optimisations carried out in construction of the prototype described in section 5.3 have introduced bugs and lead to a drop in accuracy. In order to fix those bugs, the entire network construction algorithm has been reimplmeneted.

Firstly, we have gotten rid of the `PairFunction`, introducing a new function, `sortByDistance` instead. This function sorts a given list of points by their proximity to another point:

```
distance :: Input → Input → Double
distance x y = sqrt $ sum $ map (^2) $ zipWith (-) x y

sortByDistanceTo :: Input → TrainingSet → TrainingSet
sortByDistanceTo x = sortBy distanceFunction
    where
      distanceFunction :: TrainingInput → TrainingInput → Ordering
      distanceFunction y1 y2 = compare (distance x (fst y1)) (distance x (
          ↪ fst y2))
```

Then, we amend the inner workings of `Training` as follows:

```
1  augmentNetwork :: Network → Network → TrainingInput → TrainingSet →
       ↪ Network
2  augmentNetwork outerNet n (plusOne, _) []
3      = n
4  augmentNetwork outerNet n po@(plusOne, _) ((minusOne, _):minusOnes)
5      = augmentNetwork outerNet newNet po (seave minusOnes (n `unionNet`
           ↪ newNet))
6      where
```

```
7         newNet = n `intersectNet` (hyperplane plusOne minusOne 0.5)
8
9   createNetwork' :: Network → TrainingSet → TrainingSet → TrainingSet →
        ↪ Network
10  createNetwork' n ts [] _ = n
11  createNetwork' n ts plusOnes minusOnes
12      | null $ seave ts n = n
13  createNetwork' n ts po@(plusOne:plusOnes) minusOnes
14      = createNetwork' newNet ts plusOnes minusOnes
15      where
16        newNet = n `unionNet` (augmentNetwork n emptyNet plusOne (
              ↪ sortByDistanceTo (fst plusOne) minusOnes))
17
18  createNetwork :: SeparatorFunction → TrainingSet → Network
19  createNetwork sf ts = createNetwork' startingNet ts plusOnes minusOnes
20      where
21        plusOnes = plusPoints ts
22        minusOnes = minusPoints ts
23        startingNet = sf plusOnes minusOnes
```

Let us look at the most notable differences. `augmentNetwork` now takes an additional `Network` argument, an `outerNet`, which is the network it will eventually be united with. This allows us to seave using the whole network, and not just the subnetwork for a given $+1$ point (line 5).

`createNetwork'` now takes an additional argument as well: `ts`, representing the whole training set. This allows us to consider an early stopping condition (line 12), when the current network correctly classifies the training set as a whole.

The rest of the algorithm works much as it did before, except that the $+1$ point and a list of -1 points are now passed as explicit parameters, instead of a `PairFunction` being used.

The optimisations carried out as part of this prototype have yielded a much better classification accuracy, and drastically reduced average network complexity (i.e. perceptron count). Although perceptron counts can still be quite high for non-trivial data sets, partly due to the binary nature of unions and intersections. For more detail on this, please consult section 9.

5.7 Final Prototype

The final prototype focused on implementing the ability to visualise the network produced. The UI has also been redesigned, to make it easier to use and analyse the information provided.

Network plotting

The Chart library that was used to produce the plots (as discussed in section 3.2.4) makes it simple to generate various 2D data plots. The detailed code is not complex, but repetitive, as the majority of the time is spent specifying the layout of the plots. Please consult the source code submission for details.

However, one problem we ran into is the fact that Chart can only produce 2D plots, and a lot of "real" data used for binary classification has a much higher number of attributes. The solution for those types of data sets was to plot only the projections onto each pair of axes. Note that this does not give a full picture of the network produced, as it only shows how the network behaves on the planes formed by the axes, and gives us no information regarding the space inbetween.

The way we plot the network is by approximation: we construct a set of points, with a distance of 0.1 between them, lying in the area visible on the graph produced. Then, we classify each of those points using the constructed network, and plot the results. The results are overlaid on top of the data points themselves and made semi-transparent, which allows us to clearly see misclassifications and the regions formed by combinations of separators.

```
type Projection = [Bool]
type Point2D = (Double, Double)

rangeOfNumbers :: (Double, Double) → Double → [Double]
rangeOfNumbers (from, to) step
    | from ≥ to = [to]
    | otherwise = from:(rangeOfNumbers (from+step, to) step)

samplePoints :: (Double, Double) → (Double, Double) → Double → [Point2D]
samplePoints scaleOfX scaleOfY step = [(x, y) | x ← xs, y ← ys]
    where
      xs = rangeOfNumbers scaleOfX step
      ys = rangeOfNumbers scaleOfY step

sampleSpace :: Projection → TrainingSet → [Point2D]
sampleSpace p trainingSet
    = let projectedPoints = project2D p $ map fst trainingSet
          ysc = yScale projectedPoints
          xsc = xScale projectedPoints
      in samplePoints xsc ysc sampleStep

liftPoint' :: Projection → [Double] → [Double]
liftPoint' [] _ = []
liftPoint' (_:ps) [] = 0:(liftPoint' ps [])
liftPoint' (False:ps) ds = 0:(liftPoint' ps ds)
liftPoint' (True:ps) (c:ds) = c:(liftPoint' ps ds)
```

```
liftPoint :: Projection → Point2D → [Double]
liftPoint p (x, y) = liftPoint' p [x, y]

classifyProjectedPoint :: Network → Projection → Point2D → (Point2D,
    ↪ Classification)
classifyProjectedPoint net proj point = (point, cl)
    where
      cl = runNetwork net (liftPoint proj point)
```

The function `samplePoints` generates a list of coordinates in 2-dimensional space, lying in the bounds of a rectangle specified by `scaleOfX` and `scaleOfY`, with `step` being fixed at 0.1. It does so by generating a range of numbers spanning the scale of x, another range spanning the scale of y, and producing a Cartesian product of the two ranges.

The `sampleSpace` function uses the function `xScale` and `yScale` to determine the area that will be visible and thus needs to be plotted. It uses a `Projection`, which a list of Boolean values of the same length as the rows in the data set. It should have exactly two `True` elements. The position of those elements determines which axes are used for the projection, and the function `project2D` generates the coordinates of the projected points.

The function `liftPoint` undoes the projection, filling the values at coordinates corresponding to `False` in the projection with zeros. This has the effect of making the points of the correct dimensionality to be classified by the network, but lying in the projected plane.

Finally, `classifyProjectedPoint` is a function that can be used to classify a point lying in a place, knowing the way it has been projected onto the plane. It is used by the functions generating the plots by mapping it over the lists of points generated using the `sampleSpace` function.

UI redesign

The final change made to the prototype was a redesign of the GUI to make it easier and more convenient to use. Due to the fact that Glade was used to design the interface, minimal changes needed to be done at the code level. The new layout combined the training and validation confusion matrices on a single screen, and introduced a perceptron count field to provide some form of metric of the network's complexity.

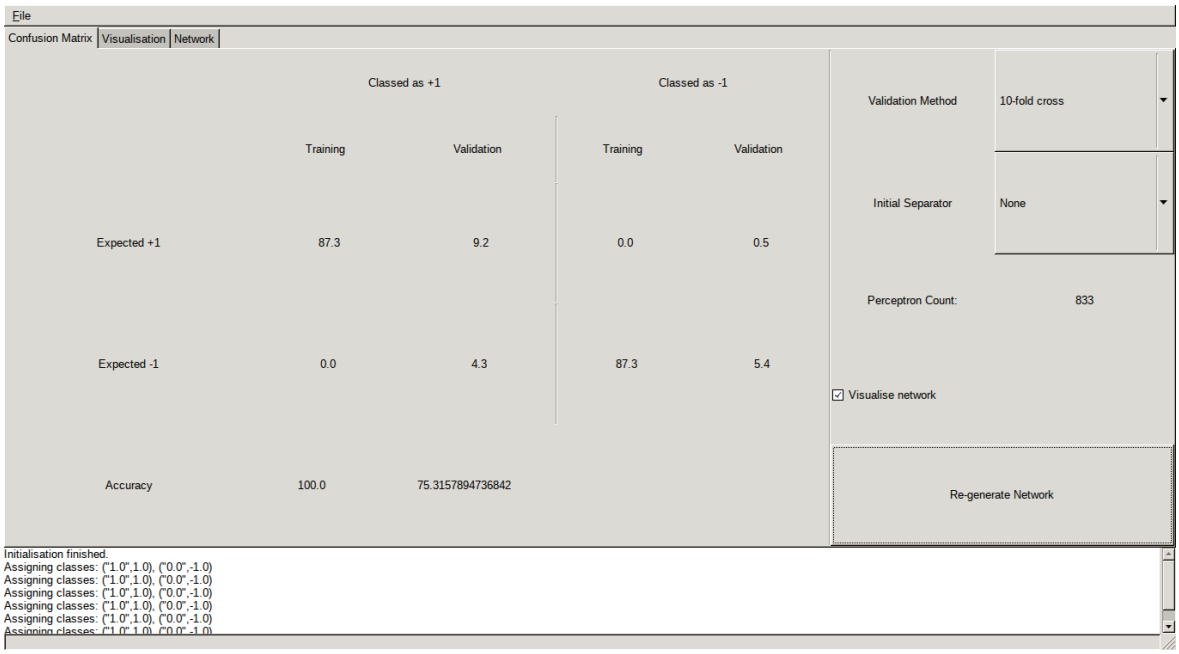


Figure 5: The main GUI screen

The file dialogue allows the user to load a new data file, and the combo boxes to the right allow to adjust the parameters, such as the validation method and the initial separator. The tick box above the "Re-generate network" button can be used to toggle visualisation on and off, as it can take a long time to visualise the network for large data sets.

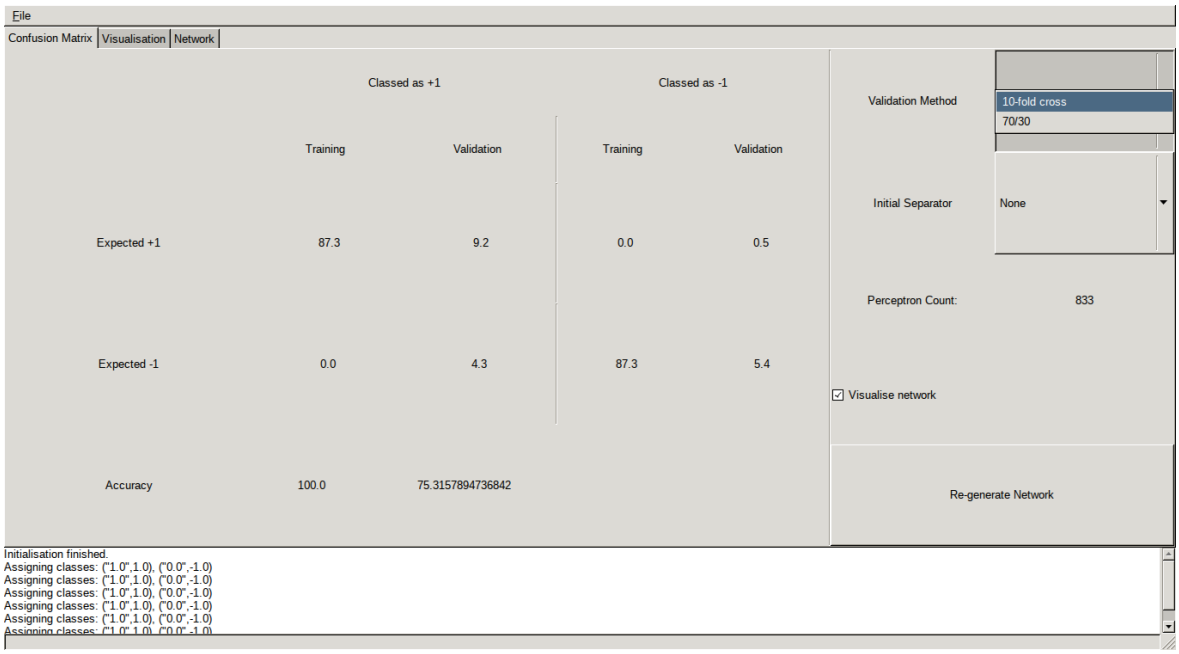


Figure 6: Choosing options

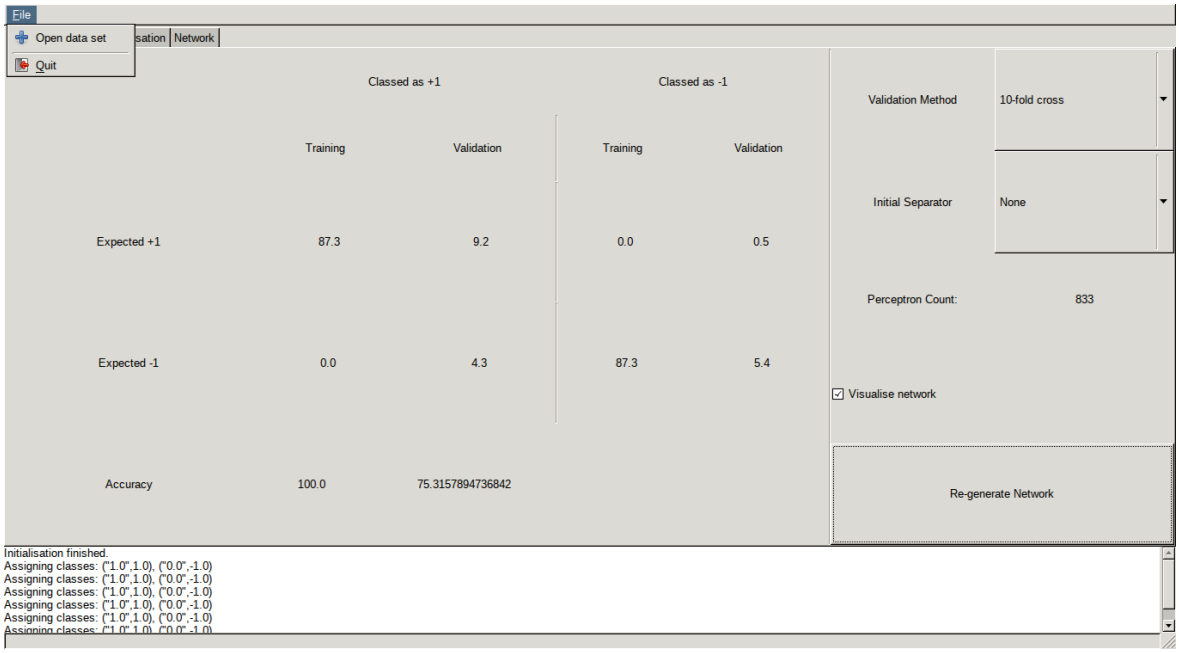


Figure 7: The file menu

The tabs at the top of the screen allow the user to navigate between the confusion matrix, the plot of a network, if one is constructed (figure 8), and the textual representation of the network (figure 9).

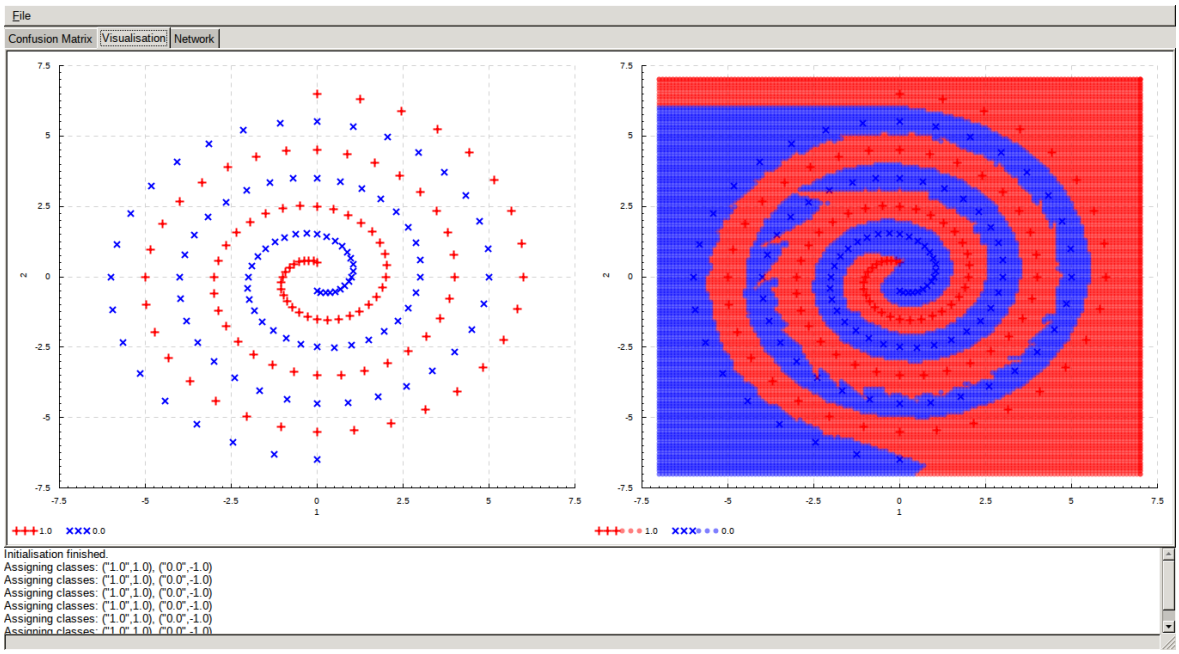


Figure 8: Visualisation tab

Figure 9: Network tab

6 Testing

Firstly, it should be mentioned that Haskell's strict typing serves as a preliminary safety net during development, eliminating several types of bugs at compile time that tend to manifest themselves as run-time bugs in less strict languages. For example, the fact that all code involving side-effects happens inside the `IO` monad [40, Ch. 7] makes it impossible for functions to have arbitrary side effects or unexpectedly mutate data. In addition, the recursive nature of Haskell programs discourages the use of shared memory, which is a known source of bugs [41].

Throughout the development of this project, the developer has taken advantage of the Glasgow Haskell Compiler's interactive mode, which allows dynamic evaluation of individual functions and examination of results, as outlined above in section 3.2.1. This helped us catch several bugs during development, before they were fully integrated into the code base.

Finally, we use QuickCheck to test certain properties of our system, as discussed in section 3.2.5, which is what the rest of this section is devoted to. As discussed in section 3.2.5, QuickCheck approaches testing by generating random data and ensuring that the properties specified by the programmer hold when the functions are applied to that data. As an example of a simple property to prove, we show that, regardless of our choice of c, $\vec{v}$ and $\vec{u}$, the hyperplane $H_c(\vec{v}, \vec{u})$ will always classify $\vec{v}$ as $+1$ and $\vec{u}$ as -1:

```
prop_HyperplaneBasic v = not (null v) ==> forAll (vectorOf (length v)
    ↪ arbitrary) $ λ(u :: [Double]) →
                        forAll (choose (1,99)) $ λ(c :: Int) →
                        (runNetwork (hyperplane v u ((fromIntegral c)/100))
                            ↪ v) == 1.0
                        && (runNetwork (hyperplane v u ((fromIntegral c)/
                            ↪ 100)) u) == (-1.0)
```

This property suggests that, for any non-empty list `v`, we can generate an arbitrary (i.e. random) list `u` of `Doubles`[2] of the same length, and an integer `c` in the range $[1, 99]$, and construct a hyperplane `hyperplane v u ((fromIntegral c)/100)`. The network corresponding to that hyperplane will classify `v` as 1.0 and `u` as -1.0, regardless of our choice of `v`, `u` and `c`.

Note that this could be proven mathematically, but such a proof would take time, and will need to be readjusted if changes are made to how we define `hyperplane` and `runNetwork`. By attempting to falsify the property on random data instead, we get good assurance that the property holds, and avoid a common pitfall of unit testing of choosing only the data we know our code can handle to test it.

In order to test some other properties networks have, we need to generate the data in a specific format. What we need is a randomly generated data set, consisting of a

[2]We make use of the ScopedTypeVariables Haskell language extension, which allows us to specify the types of arguments of anonymous functions in-line.

number of unique points with classes attached. Furthermore, we need to make sure there is at least one $+1.0$ point and at least one -1.0 point in that data set.

QuickCheck allows us to specify custom data generators using the `Gen` monad. We begin by writing a simple generator that returns two arbitrary points in n-dimensional space, represented as `[Double]`, an assign one of them a positive label, and the other a negative one:

```
twoPoints :: Int → Gen TrainingSet
twoPoints n = do
  x ← vectorOf n arbitrary
  y ← (vectorOf n arbitrary) `suchThat` (/= x)
  return [(x,1.0), (y,(-1.0))]
```

We use Haskell's do-notation here, much like it is customary to do for the `IO` monad, to make the code clearer and easier to read. It could be rewritten with no "syntactic sugar" as follows:

```
twoPoints n = vectorOf n arbitrary >>= λx →
              (vectorOf n arbitrary) `suchThat` (/= x) >>= λy →
              return [(x, 1.0), (y, (-1.0))]
```

We also note that the Haskell `return` function is not used to "return" a value from a function, but instead to lift a value into a monad (`Gen` in this case). The following examples will make use of the do-notation in order to make them easier to read, but could also just as easily be rewritten using the `>>=` (bind) operator.

We can now write a generator for m points in n-dimensional space. We make a restriction that $m \geq 2$, and simply set $m = 2$ if that is not the case, as we need at least one point belonging to each class to perform the classification.

```
mPoints :: Int → Int → Gen TrainingSet
mPoints m n
    | m < 2 = mPoints 2 n
    | otherwise = mPoints' m n []

mPoints' :: Int → Int → TrainingSet → Gen TrainingSet
mPoints' m n accum
    | m == 0 = return $ accum
    | null accum = do
  initial ← twoPoints n
  mPoints' (m-2) n initial
    | otherwise = do
    point ← (vectorOf n arbitrary) `suchThat` (λp → not $ p `elem` (map
        ↪ fst accum))
    classification ← oneof [return 1.0, return (-1.0)]
    mPoints' (m-1) n ((point,classification):accum)
```

We use accumulator-based recursion once again, and reduce `m` with each recursive call. At the start, when the accumulator is empty, we generate two random points, one with each label, using `twoPoints` above, and reduce `m` by two. In the recursive step, we generate a new point, making sure its coordinates are not already in the list (this ensures all points are unique), and assign it a class at random. This generates a list of points that can be classified by the network.

The first property we can now prove is that, by construction, a network with no initial separator should yield 100% accuracy on the data it has been trained on. We do so by constructing a network and then classifying each point in the training set with it, confirming the classes assigned are correct:

```
prop_PerfectTraining :: TrainingSet → Bool
prop_PerfectTraining ts = and $ map confirmClass ts
    where
      network = createNetwork noSeparator ts
      confirmClass (i, c) = (runNetwork network i) == c
```

Finally, we mentioned earlier in section 5.1, that an input applied to a network of perceptrons produces a classification equivalent to the one obtained by simply "running" the network using the equations of hyperplanes and combinators. Generating the network tree on each pass through construction proved to slow the training algorithm down, which prompted us to use the `runNetwork` function instead. However, we construct another property to show that a network constructed on any training data will classify any arbitrary point (possibly not in the training set) the same way, regardless of which method of applying the network is used. All the tests, at the moment, assume dimensionality 5 of the data set, but this can be easily adjusted later for more general testing:

```
prop_PerceptronEquivalence :: Property
prop_PerceptronEquivalence = forAll (vectorOf 5 arbitrary) $ λ(p :: [Double
    ↪ ]) →
                             (forAll $ mPoints 50 5) $ λts →
                             (λnet →
                              (runNetwork net p) == (Networks.classify p (
                                  ↪ perceptronNetwork net))) (createNetwork
                                  ↪ noSeparator ts)
```

All of those properties are tested 100 times, using a data set of 50 random points each time. Throughout the development process, none of the properties have been successfully falsified, giving us reasonable assurance that they hold in general.

7 Deployment

As explained in section 3.2.6, this project uses Cabal to facilitate compiling the executable file. Moreover, cabal-install can be used to place the executable and all neces-

sary files into appropriate locations in the current user's home directory. This process is outlined below.

Firstly, this process assumes that the Haskell Platform [42] is installed on the user's computer. This ensures that all the minimal necessary components (GHC, cabal-install, popular Haskell libraries) are present before compilation begins. It is easy to install and is available for a variety of platforms.

Cabal makes it very easy to install the application in the user's home directory, and then launch it. After navigating to the directory where the package has been unpacked, simply execute:

```
$ cabal install
$ ~/.cabal/bin/NeuralNetworks
```

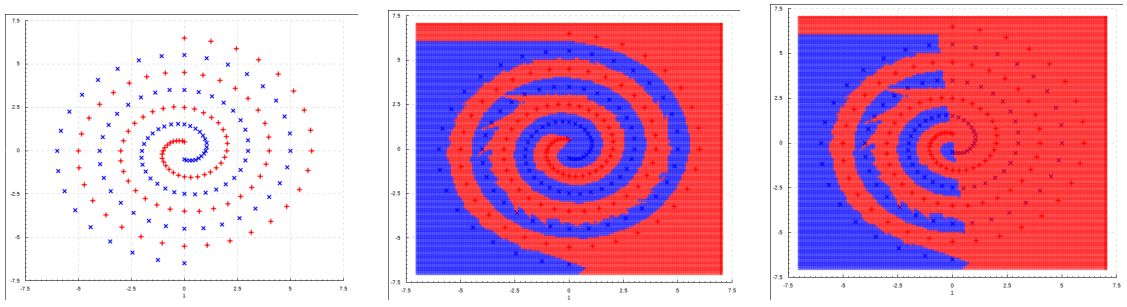
to install and launch the application.

The instructions above assume a Linux-based operating system, but should be similar for most platforms. The location of the `~/.cabal/bin` directory, however, can vary across operating systems.

Please note that the above will also attempt to fetch and compile all the dependencies, so the process can take some time to finish on the first run. In addition, it assumes that Gtk2hs can be built, which means that the appropriate development libraries and tools need to be present. Please consult the documentation relevant to your operating system [30] if problems arise.

8 Analysis of Results

Before we evaluate the performance of our network on real-world data, we turn our attention to an artificially constructed data set, described in [43]. It resembled 2 interweaving spirals, unwinding from near a common origin, and has been proposed as a benchmark for neural networks. Using the code given in the appendix of [43], we have generated the data set. The results are presented in figure 10.



(a) The spiral data set

(b) A network with no initial separator

(c) A network with the centroid initial separator

Figure 10: The network's performance on the spiral data set

As can be seen in figure 10, the network deals well with such a task, with an average accuracy of 75%. However, as can be seen in figure 10c, adding an initial separator has a dramatic impact on accuracy. Since the centres of gravity of the two classes of points nearly coincide, the separator actually offers virtually no assistance in classification, and in fact impedes network construction by misclassifying half of the points straight away. To contrast, in the absence of such a separator (figure 10b), the network follows the spirals nearly perfectly, albeit with slight "ridges" visible on the left side of the spiral.

In order to get the idea of how the resulting network performs on real-world data, we use freely available data sets provided by the "UCI Machine Learning Repository" [44]. Please note that some data sets contain missing values. In order to avoid biasing the results due to the way those missing values are treated, it has been decided to simply remove data rows that contain missing values. Moreover, certain data sets ([45,46]) had an additional field used for identification of each particular data row. Those were also removed before the data was used for training. The system assumes the classification of a data row comes as the final field. Appendix A summarises which data sets did not fulfil this condition and outlines the changes that have been made before the data was used.

The results are summarised in table 1 below. 10-fold cross-validation, discussed in more detail in section 2.6, is used to calculate the average accuracy, and the perceptron count is given for the network trained using all of the data available.

Data set	No initial separator		Centre of gravity initial separator	
	Average accuracy	Perceptron count	Average accuracy	Perceptron count
Breast Cancer Wisconsin (Diagnostic) Data Set [45]	92.23%	1501	91.95%	2635
Hepatitis Data Set [47]	81.25%	361	80.00%	363
Hepatitis Data Set [47] with Age and Sex removed	83.75%	363	82.5%	365
Parkinsons Data Set [46]	87.71%	921	84.13%	1139

Table 1: Summary of the network's performance on various data sets

From the results in table 1, we can conclude that reasonable classification accuracy can be achieved by the system produced. In examining the high perceptron count of those networks, we must note that the majority of the complexity comes from the binary nature of unions and intersections: each pair of hyperplanes must be joined separately. Thus, instead of, for example, taking an intersection of 3 hyperplanes (4 perceptrons: an intersection + 3 hyperplanes), we are producing an intersection of a hyperplane with an intersection of 2 other hyperplanes (5 perceptrons: 2 intersections + 3 hyperplanes). This will be the first thing to be addressed during future development.

We also note that sometimes removing specific attributes from a data set can increase the accuracy. For example, removing the age and gender patient information from the hepatitis data set [47] yields an increase in classification accuracy across the board. Thus more detailed analysis of data and redundant attributes might make the accuracies higher still.

Finally, we note that introducing the initial separator actually decreases the accuracy and leads to higher perceptron counts. One possible explanation is that centroids, by themselves, provide an inadequate metric to characterise the data as a whole, and therefore the rest of the network construction algorithm has to work "around" suboptimal initial separators produced.

9 Recommendations for Future Work

While the system produced as the final prototype fulfils all of the objectives posed at the beginning of this report, it is possible to further improve on the results. This section considers some of the ways in which the system can be extended to either make it more robust, more widely applicable, or easier to use.

One area of improvement that has been revealed in the section 8 is the large perceptron count (i.e. complexity) of the resulting network. A quick examination of the network produced reveals that the major cause of this complexity is the large number or union and intersection perceptrons "cascading" down to each pair of hyperplanes. If the union and intersection perceptrons were made to be non-binary, and instead able to handle arbitrary number of inputs, we could shrink the perceptron tree dramatically, achieving high reduction in network complexity.

Section 6 shows that, the current implementation, in absence of initial separators, always produces a network with 100% accuracy on training data. While this can be good in some instances, it also has the potential of making the network very prone to overfitting, which is already a concern for neural networks with a large number of data attributes, as discussed in section 2.3.4. Several possible solutions to this problem have been discussed with the academic supervisor, but none of them have been implemented at present. Potential solutions include using some sort of model to identify atypical points before network construction begins, or using clustering to identify linearly separable collections of similarly classified points and bringing those in one at a time until sufficient accuracy is achieved.

We also note that the parameter c, used to determine the position of the hyperplane in relation to the two points, is, at present, fixed at 0.5. The way of determining the most optimal value for this parameter needs evaluation, and although some possible ways of implementing it have been considered, none are present in the current iteration of the prototype. For example, the parameter could be tweaked, going from 0.5 and increasing/decreasing, considering the impact on how many new points are correctly classified with each adjustment. The value giving the best "classification gain" can then be used as the final one. This is subject to future development.

The process by which the pairs of points to be compared are chosen could also be parametarised. At present, $+1$ points are considered in sequence and -1 points are considered in order determined by their proximity to the $+1$ point under consideration. We could parametarise the function used to sort the -1 points, adding yet another drop-down list to the GUI, as well as add a similar sorting mechanism for $+1$ points. They could be shuffled, making the choice effectively random, or perhaps sorted by how close the closest minus point is to them.

Finally, currently the network is displayed to the user in text format, using simple conversion of `NetworkDesc` to `String`, as can be seen in figure 9. It would be much clearer if the network could be visualised as a perceptron diagram, although the potential complexity of such a diagram makes this a difficult task.

10 Conclusion

While there is a lot of scope for future work to be done, the final prototype produced as part of this project has achieved its primary aim of implementing and analysing a neural network construction algorithm based on simple combinations of linear classifiers. All of the objectives we have set have been achieved, and the classifiers produced by the system achieve consistent accuracies when applied to real world data.

The author of this report hopes that the refinements suggested in section 9 could be implemented in the near future, but even in its current state the system achieves its primary purpose: demonstrating the possibilities inherit within the combinations of simple, manageable linear classifiers to produce arbitrarily complex classification systems.

References

[1] A. Csenki, "A pamphlet on the construction of perceptron nets," University of Bradford, January 2015.

[2] M. J. Zaki and W. M. Jr, *Data Mining and Analysis: Fundamental Concepts and Algorithms*. New York, NY, USA: Cambridge University Press, 2014.

[3] T. Bocklitz, M. Putsche, C. Stüber, J. Käs, A. Niendorf, P. Rösch, and J. Popp, "A comprehensive study of classification methods for medical diagnosis," *Journal of Raman Spectroscopy*, vol. 40, no. 12, pp. 1759–1765, 2009.

[4] J. Liu, Y. Cao, C.-Y. Lin, Y. Huang, and M. Zhou, "Low-quality product review detection in opinion summarization." in *EMNLP-CoNLL*, 2007, pp. 334–342.

[5] Z. Yang and H. Ai, "Demographic classification with local binary patterns," in *Advances in Biometrics*. Springer, 2007, pp. 464–473.

[6] S. Marsland, *Machine Learning: An Algorithmic Perspective*, 1st ed. Chapman & Hall/CRC, 2009.

[7] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[8] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: data mining, inference and prediction*, 2nd ed. Springer, 2009.

[9] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009.

[10] S. Haykin, *Neural Networks and Learning Machines*, 3rd ed. Prentice Hall Press, 2009.

[11] P. Picton, *Neural Networks*, 2nd ed. Palgrave, 2000.

[12] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[13] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[14] B. Widrow, M. E. Hoff *et al.*, "Adaptive switching circuits." 1960.

[15] A. Csenki, 2015, private communication.

[16] D. Poole, *Linear algebra: a modern introduction*, 1st ed. Pacific Grove, CA: Brooks/Cole-Thomson Learning, 2003.

[17] K. Markham. Simple guide to confusion matrix terminology. Accessed: 25 January 2016. [Online]. Available: http://www.dataschool.io/simple-guide-to-confusion-matrix-terminology/

[18] V. S. Gordon and J. M. Bieman, "Rapid prototyping: lessons learned," *IEEE software*, no. 1, pp. 85–95, 1995.

[19] T. Gilb, "Evolutionary delivery versus the waterfall model," *ACM SIGSOFT Software Engineering Notes*, vol. 10, no. 3, pp. 49–61, 1985.

[20] B. Nuseibeh, "Weaving together requirements and architectures," *Computer*, vol. 34, no. 3, pp. 115–119, 2001.

[21] S. Balaji and M. S. Murugaiyan, "Waterfall vs. v-model vs. agile: A comparative study on sdlc," *International Journal of Information Technology and Business Management*, vol. 2, no. 1, pp. 26–30, 2012.

[22] E. A. Altameem, "Impact of agile methodology on software development," *Computer and Information Science*, vol. 8, no. 2, pp. 9–14, May 2015.

[23] Haskell implementations. Haskell Wiki. Accessed: 15 October 2015. [Online]. Available: https://wiki.haskell.org/Implementations

[24] S. Marlow and S. Peyton-Jones, "The Glasgow Haskell Compiler," in *The Architecture of Open Source Applications*, A. Brown and G. Wilson, Eds. Lulu Press, 2011, vol. 2, ch. 5, pp. 67–88.

[25] Hackage. Industrial Haskell Group. Accessed: 15 October 2015. [Online]. Available: https://hackage.haskell.org/

[26] D. Leijen, "Parsec: A parsing library for haskell," *Available online*, 2001. [Online]. Available: http://research.microsoft.com/pubs/65201/parsec-paper-letter.pdf

[27] B. O'Sullivan, J. Goerzen, and D. Stewart, *Real World Haskell*, 1st ed. O'Reilly Media, Inc., 2008.

[28] The GTK+ project. The GTK+ Team. Accessed: 20 January 2016. [Online]. Available: http://www.gtk.org/

[29] gtk: Binding to the Gtk+ graphical user interface library. Industrial Haskell Group. Accessed: 20 January 2016. [Online]. Available: https://hackage.haskell.org/package/gtk

[30] Gtk2hs/installation - haskellwiki. Haskell Wiki. Accessed: 20 January 2016. [Online]. Available: https://wiki.haskell.org/Gtk2Hs/Installation

[31] Glade - a user interface designer. The Glade project. Accessed: 20 January 2016. [Online]. Available: https://glade.gnome.org/

[32] Chart: A library for generating 2D charts and plots. Industrial Haskell Group. Accessed: 5 April 2016. [Online]. Available: https://hackage.haskell.org/package/Chart

[33] Chart-gtk: Utility functions for using the chart library with GTK. Industrial Haskell Group. Accessed: 5 April 2016. [Online]. Available: https://hackage.haskell.org/package/Chart-gtk

[34] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," *Acm sigplan notices*, vol. 46, no. 4, pp. 53–64, 2011.

[35] P. Dybjer, Q. Haiyan, and M. Takeyama, "Verifying haskell programs by combining testing and proving," in *Quality Software, 2003. Proceedings. Third International Conference on*. IEEE, 2003, pp. 272–279.

[36] ——, "Verifying haskell programs by combining testing, model checking and interactive theorem proving," *Information and software technology*, vol. 46, no. 15, pp. 1011–1025, 2004.

[37] The Haskell Cabal. common architecture for building applications and libraries. Haskell Wiki. Accessed: 15 December 2015. [Online]. Available: https://www.haskell.org/cabal/

[38] I. L. Miljenovic. Repeat after me: "Cabal is not a package manager". Accessed: 5 March 2016. [Online]. Available: https://ivanmiljenovic.wordpress.com/2010/03/15/repeat-after-me-cabal-is-not-a-package-manager/

[39] Haddock: A haskell documentation tool. Haskell Wiki. Accessed: 11 December 2015. [Online]. Available: https://www.haskell.org/haddock/

[40] S. Marlow. Haskell 2010. Language report. Accessed: 10 April 2016. [Online]. Available: https://www.haskell.org/onlinereport/haskell2010/

[41] S. Adve, "Data races are evil with no exceptions: technical perspective," *Communications of the ACM*, vol. 53, no. 11, pp. 84–84, 2010.

[42] Download Haskell Platform. Industrial Haskell Group. Accessed: 10 April 2016. [Online]. Available: https://www.haskell.org/platform/

[43] K. J. Lang and M. J. Witbrock, "Learning to tell two spirals apart," in *Proc. of 1988 Connectionist Models Summer School*, 1988.

[44] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[45] W. H. Wolberg. Breast cancer wisconsin (diagnostic) data set. University of Wisconsin Hospitals. Accessed: 10 April 2016. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29

[46] M. Little. Parkinsons data set. University of Oxford. Accessed: 10 April 2016. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/Parkinsons

[47] G. Gong. Hepatitis data set. Carnegie-Mellon University. Accessed: 10 April 2016. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Hepatitis

A Modified data sets

The hepatitis data set [47] had the class as the first parameter. It has been moved to the end of the CSV list.

The Parkinsons data set [46] had the class ("health status") as the 17th parameter, in the middle of a data row. We used Emacs' evil-mode regex-based search and replace in order to move it to the end. This is illustrated in figure 11.

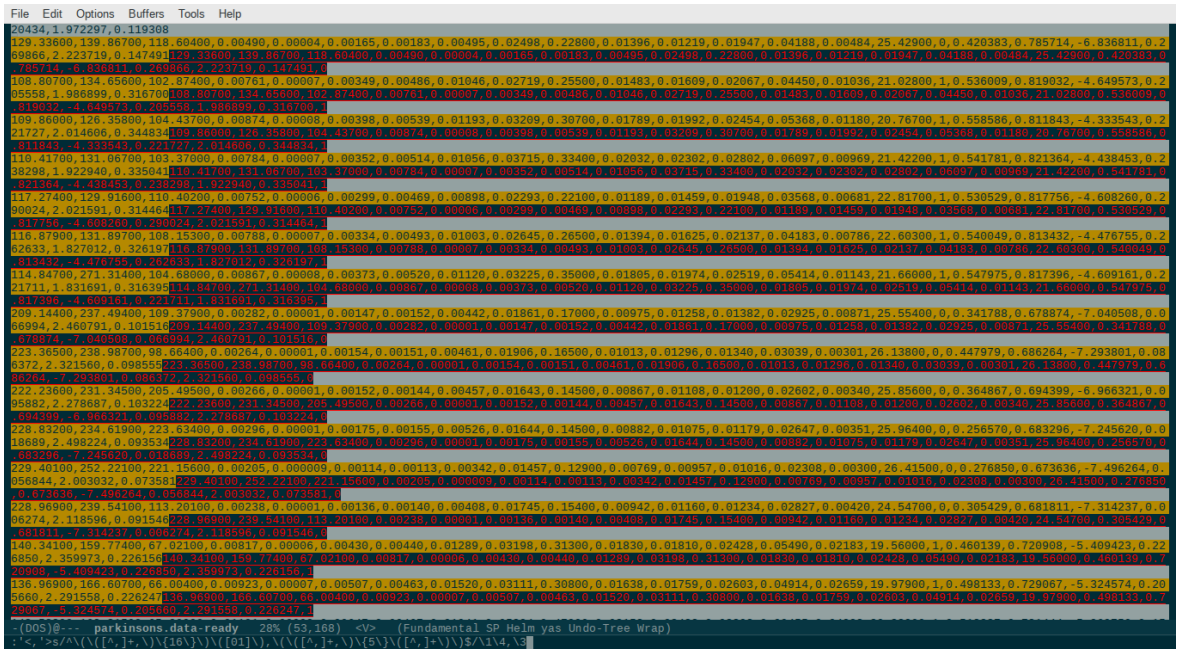


Figure 11: Using search and replace to move data columns around