10

Credit Card Transactions, Fraud Detection, and Machine Learning: Modelling Time with LSTM Recurrent Neural Networks

Bénard Wiese[1] and Christian Omlin[2]

[1] Intelligent Systems Group, Department of Computer Science
University of the Western Cape
Cape Town, South Africa
benard.wiese@gmail.com
[2] Middle East Technical University, Northern Cyprus Campus
Kalkanli, Güzelyurt, KKTC
Mersin 10, Turkey
omlin@metu.edu.tr

Abstract. In recent years, topics such as fraud detection and fraud prevention have received a lot of attention on the research front, in particular from payment card issuers. The reason for this increase in research activity can be attributed to the huge annual financial losses incurred by card issuers due to fraudulent use of their card products. A successful strategy for dealing with fraud can quite literally mean millions of dollars in savings per year on operational costs. Artificial neural networks have come to the front as an at least partially successful method for fraud detection. The success of neural networks in this field is, however, limited by their underlying design - a feedforward neural network is simply a static mapping of input vectors to output vectors, and as such is incapable of adapting to changing shopping profiles of legitimate card holders. Thus, fraud detection systems in use today are plagued by misclassifications and their usefulness is hampered by high false positive rates. We address this problem by proposing the use of a dynamic machine learning method in an attempt to model the time series inherent in sequences of same card transactions. We believe that, instead of looking at individual transactions, it makes more sense to look at sequences of transactions as a whole; a technique that can model time in this context will be more robust to minor shifts in legitimate shopping behaviour. In order to form a clear basis for comparison, we did some investigative research on feature selection, preprocessing, and on the selection of performance measures; the latter will facilitate comparison of results obtained by applying machine learning methods to the biased data sets largely associated with fraud detection. We ran experiments on real world credit card transactional data using two innovative machine learning techniques: the support vector machine (SVM) and the long short-term memory recurrent neural network (LSTM).

M. Bianchini et al. (Eds.): Innovations in Neural Infor. Paradigms & Appli., SCI 247, pp. 231–268.
springerlink.com

1 Fraud and Fraud Detection

"*Card companies continue to increase the effectiveness and sophistication of customer-profiling neural network systems that can identify at a very early stage unusual spending patterns and potentially fraudulent transactions* [6]".

There are several different factors that make payment card fraud research worthwhile. The most obvious advantage of having a proper fraud detection system in place is the restriction and control of potential monetary loss due to fraudulent activity. Annually, card issuers suffer huge financial losses due to card fraud and, consequently, large sums of money can be saved if successful and effective fraud detection techniques are applied. Ultimately, card fraud detection deals with customer behaviour profiling in the sense that each card holder exhibits an ever evolving shopping pattern; it is up to the fraud detection system to detect evident deviations from these patterns. Fraud detection is therefore a dynamic pattern recognition problem as opposed to an ordinary static binary classification problem. The question here is, given a sequence of transactions, can a classifier be used to model the time series inherent in the sequence to such an extent that deviations in card holder shopping behaviour can be detected regardless of the skewness and noise inherent in the data ? In addition our classifier must exhibit both a high probability of detection and a low false alarm rate during generalisation; otherwise, it will be practically useless. Since we are ultimately dealing with a dynamic problem here, the question also arises whether a dynamic neural network machine learning technique will outperform a static one. This study aims at answering these questions by providing a comparative analysis on the use of support vector machines and long short-term memory neural networks for payment card fraud detection.

1.1 Fraud

In the context of the payment card issuer industry, fraud can be defined as the actions undertaken by undesired elements to reap undeserved rewards, resulting in a direct monetary loss to the financial services industry. Here, we deal with the attempts by fraudsters to use stolen credit card and identity information to embezzle money, goods or services. The increase in ease of spending through the use of technology has, unfortunately, also provided a platform for increases in fraudulent activity. Fraud levels have consequently sharply risen since the 1990's, and the increase in credit card fraud is costing the payment card issuer industry literally billions of dollars annually. This has prompted the industry to come up with progressively more effective mechanisms to combat credit card fraud. More recently, the issuing industry has taken a stance to prevent fraud rather than to put mechanisms in place to minimise its effects once it takes place, and major markets have therefore taken considerable steps towards becoming EMV (Europay-Mastercard-Visa) enabled. The idea behind EMV is to use chip cards and personal identification numbers (PIN) at point of sale devices rather than authorising transactions through the use of magnetic stripes and card holder signatures. Magstriped cards have the weakness that magnetic stripes can be easily copied and reprinted on fake cards - called card skimming - and card issuers believe that chip cards, being difficult to replicate, will limit losses incurred due to card skimming. The question now is whether the necessity of fraud detection in the card issuing industry

still exists. To answer this, one has to look at the effect that EMV enablement might have on fraud patterns globally. With the shift to EMV, fraud liability will shift from the card issuers to non EMV-compliant merchants. With the onus on service establishments to ensure proper use of credit cards in their shops, a shift in fraud patterns is likely to occur. It is expected that "card-not-present" fraud will increase significantly because of chip and PIN. Card-not-present transactions take place when the physical card and card holder signature do not form part of the authorisation process, such as telephone and online purchases. Most major banks also expect ATM fraud to increase because of PIN exchange and handling in insecure environments. Card skim fraud, on the other hand, will probably migrate into countries which have not opted for EMV, such as the USA that shares borders with markets which are already EMV enabled, i.e. Canada and Mexico. Fallback fraud is also reportedly already on the increase in EMV enabled markets. Fallback happens when the chip on an EMV card is damaged and systems have to fall back on magstripe in an attempt to authorise the transaction. Some people claim that up to 45% of ATM transactions in EMV enabled markets have to fall back on magstripe, while other banks report absurd fallback figures of close to 100% in some cases. These problems are obviously due to the relative immature state of EMV as it currently stands and will probably be solved in due time; however, any expectation that this type of fraud prevention will be enough to curb credit card fraud is overly optimistic, to say the least. Payment card fraud detection is therefore, at least for now, likely to stay.

1.2 Fraud Detection

So, exactly what does fraud detection entail ? Quite simply put, fraud detection is the act of identifying fraudulent behaviour as soon as it occurs [2], which differs from fraud prevention where methods are deployed to make it increasingly more difficult for people to commit fraud in the first place. One would think that the principle of *prevention is better than cure* would also prevail here; but as discussed in the previous subsection, prevention is not always effective enough to curb the high fraud rate that plagues the payment card industry thus motivating the deployment of fraud detection mechanisms. As processing power increases, fraud detection itself might even become a prevention strategy in the future. The fact that credit cards are used in uncontrolled environments, and because legitimate card holders may only realise that they have been taken advantage of weeks after the actual fraud event, makes credit cards an easy and preferred target for fraud. A lot of money can be stolen in a very short time, leaving virtually no trace of the fraudster. The quicker fraud can be detected the better; but the large amount of data involved - sometimes thousands of transactions per second - makes real-time detection difficult and sometimes even infeasible.

The development of new fraud detection systems is hampered by the limitation of the exchange of ideas pertaining to this subject, simply because it does not make sense to describe fraud detection and prevention techniques in great detail in the public domain [2]. Making details of fraud detection techniques public will give fraudsters exactly what they need to devise strategies to beat these systems. Fraud, especially in the context of the financial services industry, is seen as a very sensitive topic because of the stigma attached to potential monetary loss. Card issuers are therefore usually very hesitant to report annual fraud figures and better fraud detection strategies or systems

than those of the competition are therefore advantageous; this gives even more reason for issuers to keep internal research results on fraud detection away from the public domain. Data sets and results are therefore seldom made public, making it difficult to assess the effectiveness of new fraud detection strategies and techniques. Most fraud detection systems today are awkward to use and become ineffective with time because of changing shopping behaviour and migrating fraud patterns. During the holiday seasons, for example, shopping profiles change significantly and fraud detection systems cannot deal with changing behaviour because of their static nature and inability to dynamically adapt to changes in patterns. Fraud detection systems therefore suffer from unacceptable false alarm rates, making the probability of annoying legitimate customers much higher than that of actually detecting fraud. Systems based on Hidden Markov Models promise better results, but are useless for real-time use when transaction volumes become too high.

In this chapter, we investigate the use of two techniques that are neural network based and therefore promises relative quick classification times, of which one is also dynamic because it attempts to learn the underlying time series present in series of same card holder transactions. These two methodologies, namely the support vector machine (SVM) and the long short-term memory neural network (LSTM), are discussed in the next two sections.

2 Methodology I: Support Vector Machines

In 1992, Boser, Guyon and Vapnik proposed a training algorithm for optimal margin classifiers in which they showed that maximising the margin between training examples and class boundary amounts to minimising the maximum loss with regards to the generalisation performance of the classifier [3]. This idea was initially explored because binary class optimal margin classifiers achieve errorless separation of the training data, given that separation is possible, and outliers are easily identified in such a classifier. The first investigations into this type of algorithm were based on separable data sets, but, in 1995, Cortes and Vapnik extended the algorithm to account for linearly inseparable data [9]; attempts soon followed to also extend the results to multi-class classification problems. This type of learning machine was later dubbed the support vector machine (SVM). The SVM is a machine learning technique with a strong and sound theoretical basis. It is interesting to note that, in most cases, researchers claim that SVMs match or outperform neural networks in classification problems. In this chapter we will put these claims to the test on a non-trivial, real world problem. We assume the readers to be familiar with the intricacies of SVMs, and we therefore only offer a succinct discussion here for the sake of completeness.

2.1 The Maximum Margin Hyperplane

During the supervised training process of a classifier, training vectors or patterns of the form $(\mathbf{x},y)$, where $\mathbf{x}$ is a set of input parameters or features and y denotes class membership, are repeatedly presented to the classifier in an attempt to learn a decision function $D(\mathbf{x})$, which can later be used to make classification decisions on previously unseen

data. In the case of the optimal margin training algorithm, these decision functions have to be linear in their parameters, but are not restricted to linear dependencies in their input components $\boldsymbol{x}$, and can also be expressed in either direct or dual space [3]. In direct space, the decision function has the following form:

$$D(\mathbf{x}) = \sum_{i=1}^{N} w_i \varphi_i(\mathbf{x}) + b \tag{1}$$

This is identical to the original perceptron decision function from which the first multilayer networks were derived, with the bias represented by b instead of w_0 and φ_i some function of $\mathbf{x}$. In the perceptron, φ_i is simply the identity function. In the case of a binary outcome decision function applied to linearly separable data, the function can be represented in 2-dimensional space as a straight line splitting the input vectors into the two classes they might possibly belong to, as demonstrated in figure 1.

In its simplest form, the SVM algorithm will construct a hyperplane that completely separates (at least in the linear separable case) the data in such a way that $\mathbf{w} \cdot \mathbf{x} + b > 0$ for all points belonging to one class, and $\mathbf{w} \cdot \mathbf{x} + b < 0$ for all points in the other class [19]. In other words, for $D(\mathbf{x}) > 0$ pattern $\mathbf{x}$ belongs to class A and for $D(\mathbf{x}) < 0$ pattern $\mathbf{x}$ belongs to class B. In this context, $D(\mathbf{x})$ is referred to as the decision surface or separating hyperplane and all points $\mathbf{x}$ which lie on this decision surface satisfy the equation $\mathbf{w} \cdot \mathbf{x} + b = 0$.

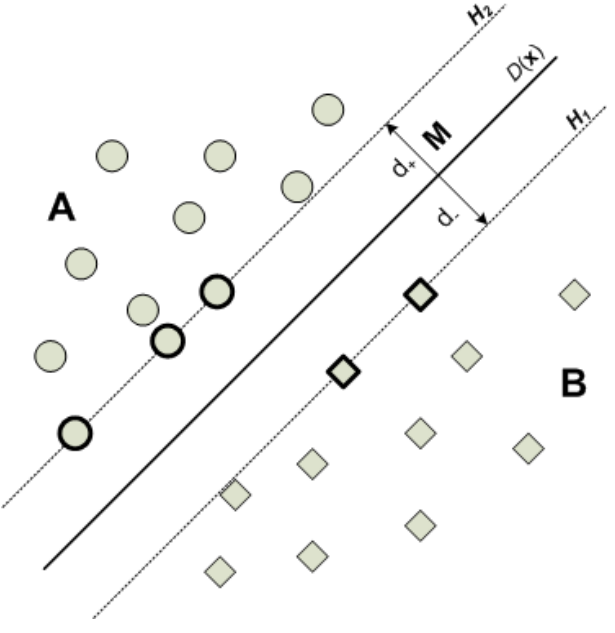

Fig. 1. A binary outcome decision function in 2-dimensional space (adapted from [5]). The support vectors are shown with extra thick borders, with *H1* and *H2* denoting the separating hyperplanes on which they lie. The maximum margin is denoted as **M** and gives the maximum distance between the separating hyperplanes

Consider that the two possible classes to which an arbitrary pattern can belong are identified by labels y_i, where $y_i \in \{-1, 1\}$. Furthermore, define $d_+(d_-)$ to be the shortest distance between the separating hyperplane and the closest positive (negative) pattern in the training set. We can then define the margin (denoted $\boldsymbol{M}$ in figure 1) to be $d_+(d_-)$, a quantity that the support vector algorithm will attempt to maximise during training. In the linear separable case, all training data will adhere to the following two constraints:

$$w \cdot \mathbf{x}_i + b \geq +1 \text{ (for } y_i = +1) \tag{2}$$

$$w \cdot \mathbf{x}_i + b \leq -1 \text{ (for } y_i = -1) \tag{3}$$

A vector $\mathbf{w}$ and scalar b therefore exist such that

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \geq 0,\ i \in \{1, ..., l\} \tag{4}$$

The points for which the equalities in equations (2) and (3) hold lie on two separate hyperplanes parallel to the separating hyperplane, but on different sides as shown by H_1 and H_2 in figure 1. These hyperplanes can be defined as H_1:$\mathbf{w} \cdot \mathbf{x}_i + b = 1$ and H_2: $\mathbf{w} \cdot \mathbf{x}_i + b = -1$, where $\mathbf{w}$ is normal to the hyperplanes in both cases. The perpendicular distance between H_1 and the origin and H_2 and the origin is therefore $\frac{|1-b|}{\|\mathbf{w}\|}$ and $\frac{|-1-b|}{\|\mathbf{w}\|}$, respectively, and it then follows from this that $d_+ = d_- = \frac{1}{\|\mathbf{w}\|}$ and margin $\mathbf{M} = \frac{2}{\|\mathbf{w}\|}$.

A support vector can now be defined as a training pattern which lies on either H_1 or H_2, and whose removal might change the size of the maximum margin. In figure 1, the support vectors are shown as circles or squares with extra thick borders. We can find the set of hyperplanes H_1 and H_2 that gives the maximum margin by maximising the quantity $\frac{1}{\|\mathbf{w}\|}$ (or likewise minimising $\frac{1}{2}\|\mathbf{w}\|^2$). This is equivalent to solving the quadratic problem

$$\min_{\mathbf{w}} \|\mathbf{w}\|^2 \tag{5}$$

under the constraints in (4). Although the quadratic problem in (5) can be solved directly with numerical techniques, this approach becomes impractical when the dimensionality of the φ-space becomes large; furthermore, no information about the support vectors is gained when solving the problem in direct space [3]. These problems are addressed by transforming the problem in (5) from direct space into dual space.

2.2 The Soft Margin Hyperplane

The main assumption under which all of the equations in the previous section are derived is that the training data is linearly separable. This is, however, a severe oversimplification of real world data sets and the SVM algorithm will have little to no practical value if not extended to handle linearly inseparable problems. In feedforward neural networks, inseparable training sets with high dimensionality are normally sufficiently handled by minimising some training error in the context of some predefined error measure. The same principle can be applied to SVMs by introducing positive variables $\xi_i \geq 0,\ i = 1, ..., l$ so that we have the following constraints

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i,\ i \in \{1, ..., l\} \tag{6}$$

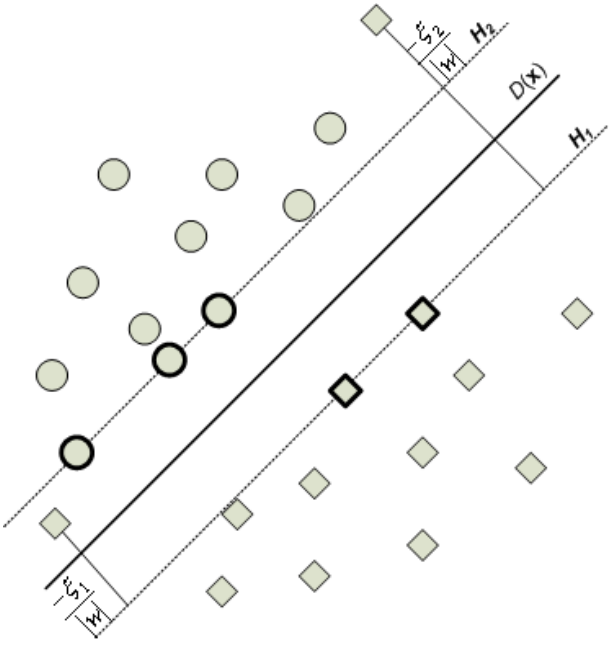

Fig. 2. Linear separating hyperplanes for the linearly inseparable case (adapted from [5]). Data vectors on the wrong side of the decision surface are compensated for by the introduction of slack variables, whose magnitude need to exceed unity for an error to occur

$$\xi \geq 0, \ \forall i \tag{7}$$

The positive variables ξ_i in (6) and (7) are called slack variables; they allow for some margin of error (i.e. slack) when deciding on which side of the optimal hyperplane a training exemplar lies. In fact, ξ_i must exceed unity for an error to occur and $\sum_i \xi_i$ represents an upper bound on the number of training errors [9][5]. The introduction of slack variables relaxes the original constraints in (4) somewhat. During training, we would like the separation error (and hence the ξ_i's) to be as small as possible and we therefore introduce a penalty on the objective function by changing it to

$$\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{l}\xi_i \tag{8}$$

where $C \in \Re$ is a cost parameter that represents the cost of making an error, or the extent to which violations of the original constraints (4) should be penalised [19]. There is no straightforward method for selecting the value of the cost parameter C, but a number of techniques exist; the most popular method being grid-search [15] using cross-validation [9]. Figure 2 depicts the use of slack variables.

Summary. The support vector machine classifier method is a relatively simple technique with a strong theoretical and mathematical basis, and most researchers claim that it exhibits above average performance when compared to neural network methods [3][5][9].

In this chapter we will empirically compare the SVM to another novel machine learning technique, the long short-term memory recurrent neural network, discussed in the next section.

3 Methodology II: Long Short-Term Recurrent Neural Networks

The current state in a recurrent network depends on all previous time steps of a particular classification attempt. Most RNN learning algorithms fail on problems with long minimum time lags between input signals and their resulting error signals. When an error signal is injected back into a network - and hence flowing back in time - it tends to either blow up or vanish. An error that blows up leads to oscillating weights and unpredicted network behaviour, while a vanishing error will quite obviously cause training to slow down or completely stop during a network's attempts to learn to bridge long time lags.

Methods like Time-Delay Neural Networks (Lang et al. 1990), Plate's method (Plate 1993) and NARX networks (Lin et al. 1995) are examples of learning algorithms that are applicable to short time gap problems only. These will, in all probability, not perform very well on our specific problem. Many attempts have been made to address the long time lag problem. Bengio et al. (1994) investigated a range of alternatives to gradient based methods, which included simulated annealing, multi-grid random search, time-weighted pseudo-Newton optimisation, and discrete error propagation. Simulated annealing performed the best on all their experimental problems, but it requires a lot more training time than the others, with training time also increasing as sequence length increases. As reported in [14], most of these attempts to bridge long time lags can be outperformed by simple random weight guessing. Other methods, for example unit addition (Ring 1993) and Kalman Filter RNNs (Puskouius and Feldkamp 1994), promises good results on longer time lags, but these are not applicable to problems with unspecified lag durations.

We discuss here the problem of vanishing gradients during training in more detail, along with a solution to the problem later dubbed Long Short-Term Memory: a novel machine learning technique that form the basis for our second methodology for payment card fraud detection.

3.1 The Vanishing Gradient Problem

To gain a better understanding of why the error tends to suffer from exponential decay over long time lags, it is worthwhile to take a look at Hochreiter's analysis [14]. Apply BPTT to a fully recurrent network whose non-input indices range from 1 to n. The error signal computed at an arbitrary unit a at time step t is propagated back through time for q time steps to arbitrary unit b, causing the error to be scaled by the factor

$$\frac{\partial \delta_b(t-q)}{\partial \delta_a(t)} = \begin{cases} f_b'(net_b(t-1))w_{ab} & q = 1 \\ f_b'(net_b(t-q)) \sum_{i=1}^{n} \frac{\partial \delta_i(t-q+1)}{\partial \delta_a(t)} w_{ib} & q > 1 \end{cases}. \tag{9}$$

Setting $i_0 = a$ and $i_q = b$, one can derive the following equation through proof by induction:

$$\frac{\partial \delta_b(t-q)}{\partial \delta_a(t)} = \sum_{i_1=1}^{n} \cdots \sum_{i_q=1}^{n} \prod_{j=1}^{q} f_{i_j}'(net_{i_j}(t-j)) w_{i_j i_{j-1}} \tag{10}$$

where the sum of the n^{q-1} terms, $\prod_{j=1}^{q} f'_{i_j}(net_{i_j}(t-j))w_{i_j i_{j-1}}$, represents the total error flowing back from unit a to unit b.

It then follows that, if

$$f'_{i_j}(net_{i_j}(t-j))w_{i_j i_{j-1}} > 1 \tag{11}$$

for all j, then the product term in equation (10) will grow exponentially with increasing q. In other words the error will blow up and conflicting error signals arriving at unit b can lead to oscillating weights and unstable learning [14]. On the other hand, if

$$f'_{i_j}(net_{i_j}(t-j))w_{i_j i_{j-1}} < 1 \tag{12}$$

for all j, then the product term will decrease exponentially with q and the error will vanish, making it impossible for the network to learn anything or slowing learning down to an unacceptable speed.

3.2 Long Short-Term Memory

So how do we avoid vanishing error signals in recurrent neural networks ? The most obvious approach would be to ensure constant error flow through each unit in the network. The implementation of such a learning method, however, might not be as obvious or straightforward. Hochreiter and Schmidhuber addressed the issue of constant error flow in [14] by introducing a novel machine learning method they dubbed long short-term memory (LSTM).

The Constant Error Carrousel. Concentrating on a single unit j and looking at one time step, it follows from Equation (9) that j's local error back flow at time t is given by $\delta_j(t) = f'_j(net_j(t))\delta_j(t+1)w_{jj}$, with w_{jj} the weight which connects the unit to itself. It is evident from Equations (11) and (12) that, in order to enforce constant error flow through unit j, we need to have

$$f'_j(net_j(t))w_{jj} = 1.$$

Integrating the equation above gives

$$\begin{aligned} &\int \partial f_j(net_j(t)) = \int \tfrac{1}{w_{jj}} \partial_{net_j} \\ &\therefore f_j(net_j(t)) = \tfrac{net_j(t)}{w_{jj}}. \end{aligned}$$

We learn two details from the above equation:

1. f_j has to be linear; and
2. unit j's activation has to remain constant, i.e.
   $y_j(t+1) = f_j(net_j(t+1)) = f_j(w_{jj}y_j(t)) = y_j(t)$.

This is achieved by using the identity function $f_j : f_j(x) = x, \forall x$, and by setting $w_{jj} = 1$. Hochreiter and Schmidhuber refer to this as the constant error carrousel (CEC). The CEC performs a memorising function, or to be more precise, it is the device through which short-term memory storage is achieved for extended periods of time in an LSTM network.

Input and Output Gates. The previous subsection introduced the CEC as an arbitrary unit j with a single connection back to itself. Since this arbitrary unit j will also be connected to other units in the network, the effect that weighted inputs have on the unit has to be taken into account. Likewise, the effect that unit j's weighted outputs have on other units in the network has to be closely scrutinised. In the case of incoming weighted connections to unit j, it is quite possible for these weights to receive conflicting update signals during training, which in turn makes learning difficult because the same weight is used to store certain inputs and ignore others [14]. In the same way, output weights originating at unit j can receive conflicting weight update signals because the same weights can be used to retrieve j's contents sometimes, and prevent j's output to flow forward through the network at other times. The problem of conflicting update signals for input and output weights is addressed in the LSTM architecture with the introduction of multiplicative input and output gates. The input gate of a memory cell is taught when to open and when to close, thereby controlling when network inputs to a memory cell are allowed to adjust its memory contents; an input gate therefore also helps to protect the memory cell contents from being disturbed by irrelevant input from other network units or memory cells. In the same way, an output gate is taught when to open and close, thereby controlling access to the memory cell's contents. An output gate therefore helps to protect the memory contents of other memory cells from being disturbed by irrelevant output from its memory cell.

The LSTM Memory Block. An LSTM network unit containing a CEC, an input gate, and an output gate, is called a *memory cell*. Figure 3 depicts the standard architecture of an LSTM memory cell. One or more memory cells that share input and output gates between them are grouped together in a unit called the *memory block*. Each cell in a memory block has its own CEC at its core to ensure constant error flow through the cell in the absence of input or error signals, thereby solving the vanishing gradient problem. The activation of a CEC is called the internal cell state [11].

Forward Pass with LSTM. Borrowing from [11], we define the following indexes: j indexes memory blocks; v indexes memory cells in block j such that c_j^v denotes the v-th cell of the j-th memory block; w_{lm} denotes a weight connecting unit m to unit l; and m indexes source units as applicable. A memory cell has three major sets of inputs: standard cell input, input gate input and output gate input. The cell input to arbitrary memory cell c_j^v at time t is

$$net_{c_j^v}(t) = \sum_m w_{c_j^v m} y_m(t-1). \tag{13}$$

The input gate activation y_{in} is

$$y_{in_j} = f_{in_j}\left(\sum_m w_{in_j m} y_m(t-1)\right) \tag{14}$$

while output gate activation is computed as

$$y_{out_j} = f_{out_j}\left(\sum_m w_{out_j m} y_m(t-1)\right). \tag{15}$$

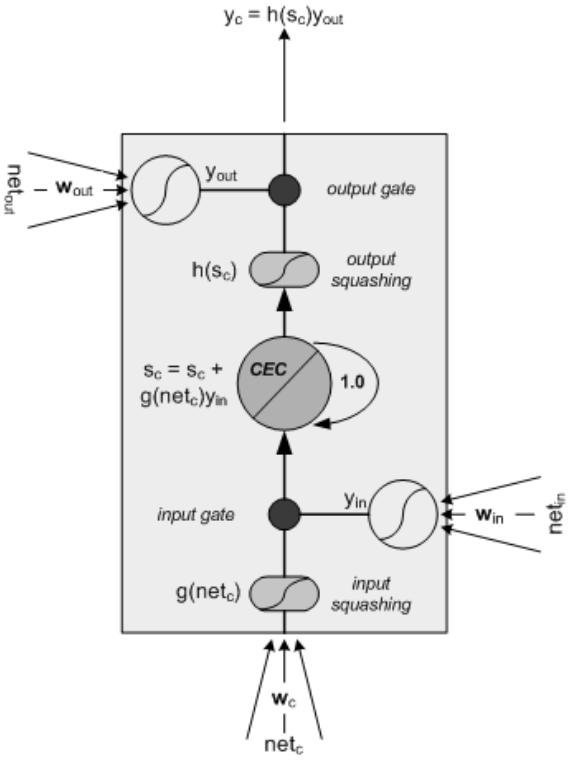

Fig. 3. An LSTM memory cell

The squashing function f used in the gates is a standard logistic sigmoid with output range [0,1]:

$$f(x) = \frac{1}{1+e^{-x}}. \tag{16}$$

Assuming that the internal state of a memory cell at time $t = 0$ is $s_{c_j^v}(0) = 0$, the internal state of memory cell c at time t is calculated by adding the squashed, gated input to the internal cell state of the last time step, $s_c(t-1)$:

$$s_{c_j^v}(t) = s_{c_j^v}(t-1) + y_{in_j}(t)g(net_{c_j^v}(t)) \quad \text{for } t > 0, \tag{17}$$

where the input squashing function g is given by [14]

$$g(x) = \frac{4}{1+e^{-x}} - 2. \tag{18}$$

The cell output y^c can then be calculated by squashing the internal state s_c and multiplying the result by the output gate activation,

$$y_{c_j^v}(t) = y_{out_j}(t)h(s_{c_j^v}(t)), \tag{19}$$

where the output squashing function h is given by

$$h(x) = \frac{2}{1+e^{-x}} - 1. \tag{20}$$

It was later suggested that the output squashing function h can be removed from equation (19) because no empirical evidence exists that it is needed [12]. Equation (19) is thus changed to

$$y_{c_j^v}(t) = y_{out_j}(t) s_{c_j^v}(t). \tag{21}$$

Finally, the activations of the output units (indexed by k) can be calculated as

$$y_k(t) = f_k(net_k(t)),\ net_k(t) = \sum_m w_{km} y_m(t-1). \tag{22}$$

Forget Gates. The standard LSTM architecture described above, although powerful, has its limitations. According to [11], the cell state s_c often tends to grow linearly during presentation of a time series, which might lead to saturation of the output squashing function h if the network is presented with a continuous input stream. Saturation of h will reduce the LSTM memory cell to a regular BPTT unit, causing it to lose its ability to memorise. Furthermore, saturation of h will cause its derivative to vanish, which will in turn block any error signals from entering the cell during back propagation. Cell state growth can be limited by manually resetting the state at the start of each new sequence. This method, however, is not practical in cases where the sequence has no discernable end, or where no external teacher exists for subdividing the input sequence into sub sequences. A sequence of credit card transactions is one such an example.

Gers, Schmidhuber and Cummins [11] solved the cell state saturation problem by introducing a third gate into the LSTM memory block architecture. This third gate is designed to learn to reset cell states when their content becomes useless to the network. In a way it forces the cell to forget what it had memorised during earlier time steps, and is therefore called a *forget gate*. Figure 4 shows the extended LSTM memory cell with a forget gate. The only change that the addition of a forget gate introduces to the forward pass of LSTM is in Equation (17), where the squashed, gated cell input is now added to the forget gated cell state of the previous time step, instead of just the basic cell state:

$$s_{c_j^v}(t) = y_{\varphi_j} s_{c_j^v}(t-1) + y_{in_j}(t) g(net_{c_j^v}(t)) \text{ for } t > 0, \tag{23}$$

where y_φ is the forget gate activation and is calculated (like the activations of the other gates) by squashing the weighted input to the gate:

$$y_{\varphi_j} = f_{\varphi_j}(\sum_m w_{\varphi_j m} y_m(t-1)). \tag{24}$$

Once again, f_φ is the standard logistic function of equation (16). It was suggested in [11] to initialise input and output gate weights with negative values, and forget gate weights with positive values, in order to ensure that a memory cell behaves like a normal LSTM cell during the beginning phases of training, as not to forget anything until it has learned to do so.

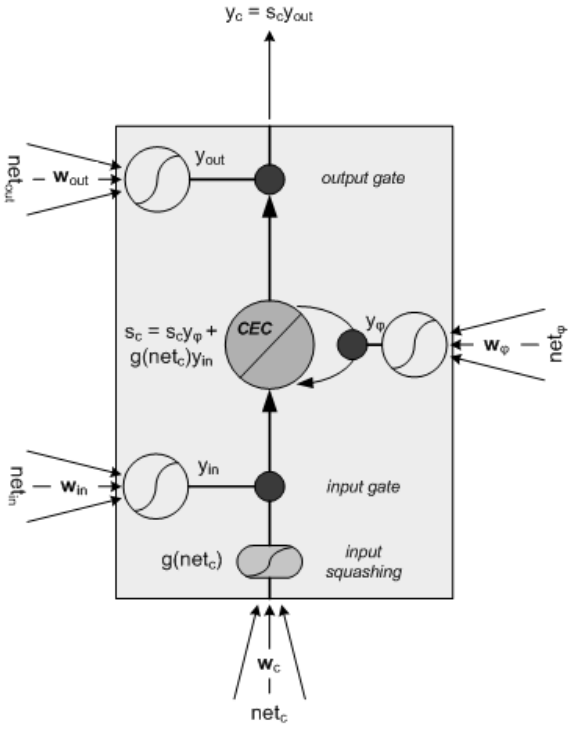

Fig. 4. An LSTM memory cell with a forget gate

Peephole Connections. Another addition to the LSTM memory cell architecture is the notion of peephole connections, first introduced by Gers, Schraudolph and Schmidhuber in 2002 [12]. Peephole connections basically connect a memory cell CEC with the memory block gates through additional waited connections. This was done to give the gates some feedback on the internal cell states which they are suppose to control. Peephole connections are not used in any of the learning tasks studied in this chapter; their use in initial experiments did not show significant improvement in generalisation performance and for the sake of simplicity further investigations with peephole connections were therefore discontinued. The interested reader can get more information on peephole connections and the resulting modified LSTM forward pass in [12].

LSTM Training - A Hybrid Approach. The LSTM backward pass makes use of truncated versions of both the BPTT and RTRL algorithms to calculate the weight deltas for a particular time step. This hybrid approach to RNN learning was first suggested by Williams (1989) and later described by Schmidhuber (1992). In the context of LSTM learning and the BPTT and RTRL algorithms, truncation means that errors are cut off once they leak out of a memory cell or gate although they do serve to change incoming weights [11].

The standard BPTT algorithm is used to calculate the weight deltas for output unit weights while truncated BPTT is used for output gate weights. A truncated version

of RTRL is used to calculate deltas for cell, input gate, and forget gate weights. The rationale behind truncation is that it makes learning computationally feasible without significantly decreasing network training and generalisation performance. Detailed descriptions of LSTM training can be found in [11], [12] or [14].

Summary. This section dealt explicitly with recurrent neural networks as a possible solution to the problem of detecting fraudulent credit card transactions. The motivation behind this is that recurrent neural network training is a tried and tested technique for dealing with sequence classification problems, of which the classification of chains of time ordered credit card transactions is a perfect real world example. A serious shortcoming of recurrent neural networks, namely the vanishing gradient problem, was highlighted and a possible solution in the form of LSTM recurrent neural networks [14] was described. Here we are interested in modelling very long time series, and in [14] and [11] it was shown that LSTM, in particular, is exceedingly good at modelling long-term dependencies in sequence classification tasks.

Ultimately, this chapter will compare the performance of support vector machines with that of long short-term memory recurrent neural networks as applied to the non-trivial problem of fraud transaction detection. However, before this can be done a criteria for evaluating the performance of each methodology needs to be selected. The next section looks at the performance evaluation of these two machine learning techniques as applied to a binary classification problem in more detail.

4 Evaluating Performance

This section introduces two techniques that will be used to evaluate the generalisation performance of the networks used in the experiments in section 6. The popular mean squared error is discussed first along with its advantages and disadvantages. It is also shown why the mean squared error is not a good measure of performance when dealing with skew data sets and binary classifiers. The discussion on mean squared error is followed by a discussion of an alternative performance measure that is more applicable to fraud detection and binary classifiers in general - the receiver operating characteristic curve.

4.1 Mean Squared Error

The mean squared error (MSE) is probably the most common error measure used in both the training of a wide range of neural networks, and the calculation of neural network generalisation performance measures. The MSE of a neural network is calculated by simply summing the sum of the square of the difference between the actual output and expected output values of the network, and dividing the result by the number of outputs n and training exemplars m:

$$MSE_{net} = \frac{1}{mn} \sum_{d} \sum_{k} (d_k - y_k)^2 \tag{25}$$

As usual, indices k denotes output neurons.

When used for training, the factor $1/n$ in equation (25) is often simplified to 1/2 in order to make subsequent calculations more elegant[1]. The ease with which MSE's derivative can be calculated is one of the reasons for its popularity in the machine learning field. Other reasons include the fact that MSE emphasise large errors more than smaller ones. In contrast to the positive features of MSE, it remains a pure mathematical construct which fails to take the cost of classification errors into account. Furthermore, it also fails to clearly distinguish minor errors from serious errors. When dealing with skewed data sets which exhibit distributions heavily in favour of a particular class, a performance measure obtained using MSE might be misleading or even nonsensical. Take for example a credit card transaction data set containing data divided into two classes: class **A** representing legitimate transactions and class **B** representing fraudulent transactions. As is normal with transactional data sets, it is expected that the occurrence of class **A** will be in the majority by a large margin, possibly representing in excess of 99.9% of the data set with class **B** representing a mere 0.01%. Let us further assume that we have a really simple classifier that constantly labels input vectors as belonging to class **A**, regardless of the content of the vectors. Also assume that the output squashing function always rounds the output of the classifier to 0.9 (to represent class **A**). In such a case the success rate of the classifier will be 99.9% and its MSE will be negligible; but in spite of this "brilliant" generalisation performance, the classifier would not have identified one single fraudulent transaction and will hence be completely useless. If we go further and attach a cost to misclassifying fraudulent transactions, the shortcomings of MSE becomes even more apparent. The next subsection introduces an alternative to MSE which is particularly useful in measuring the performance of binary classifiers.

4.2 Receiver Operating Characteristic

In the 1950's, a major theoretical advance was made by combining detection theory with statistical decision theory, and the importance of measuring two aspects of detection performance was realised, namely that one must measure the conditional probability that the observer decides the condition is present, the so-called hit rate, or that the observer decides the condition is present when it is actually not, called the false alarm rate. This can be represented in the format shown in table 1, called the outcome probability table. This table is often used to visualise the probability of the four possible classification outcomes of binary classifiers, or in our case a fraud detection system. In table 1 the rows represent the actual labels [legitimate,fraud] that an arbitrary transaction might have, while the columns represent the labels that a classifier might assign to such a transaction.

As stated in [4], a high correct classification probability is represented by

$$P(correct) = P(correct|fraud)P(fraud) + P(correct|legal)P(legal). \quad (26)$$

Put differently, equation (26) simply states that the classifier will be at its best when it succeeds in maximising both the number of correctly classified fraudulent transactions

[1] The error gradient information for weights is calculated by differentiating the error measure with regards to the weights, which will ultimately make the factor 1/2 vanish.

Table 1. The outcome probability table [4]

Assigned/Actual	Legitimate	fraud
legitimate	P(correct|legal)	P(false alarm|legal)
fraud	P(fraud not detected)	P(correct|fraud)

and correctly classified legal transactions. This is obviously achieved by minimising the number of misclassifications, or in other words minimising the weighted sum

$$C = c_1 P(fraud\ not\ detected) + c_2 P(false\ alarm|legitimate). \tag{27}$$

Since it is difficult to determine c_1 and c_2 in practice, it makes more sense to attempt to maximise the number of fraudulent transactions detected while minimising the false alarm rate. Table 2 shows a simplified version of the classification outcome matrix containing descriptions and abbreviations of the four possible outcomes of a classification with a binary target class. These abbreviations and terminology will be used in the derivations to follow.

Table 2. A simplified outcome/decision matrix

Assigned/Actual	legitimate	fraud
legitimate	True Negative (TN)	False Positive (FP)
fraud	False Negative (FN)	True Positive (TP)

In binary classification problems, a classifier simply attempts to predict whether a given condition is present or not for each item in a data set. In our case, it attempts to predict whether a transaction is genuine or fraudulent, or whether, given a range of earlier transactions, the next transaction in a sequence is legitimate or not. Decisions on whether a condition is present or absent are, in most cases, based on the activation level that a single output neuron achieves during classification. A low activation points towards the condition being absent, while a high activation level indicates that the condition is indeed present. Two types of errors are possible with the decision scheme set out in table 2. A *type I* error, or false positive, is made when the classifier decides that the condition is present when it is not, and a *type II* error, or false negative, is made when the classifier decides that the condition is absent when it is in fact present. In order to distinguish between high and low activations in the continuous activation spectrum of a single output neuron, a threshold is normally chosen and any activation lower than the threshold is then perceived as an indication that the condition in question is absent, while any activation equal to or above the threshold suggests that the condition is probably present. Figure 5 illustrates the high and low activation probabilities of an arbitrary output neuron. In statistics, the curve on the left representing the probability that the condition is absent is called the null hypothesis, while the curve on the right measuring the probability of the condition being present is called the alternative hypothesis [22]. It is important to note that Figure 5 shows a significant overlap between the null and alternative output probabilities, i.e. a somewhat grey decision area where the decision outcome is not certain beyond any doubt. This is usually the case in real world problems where perfect performance is unachievable.

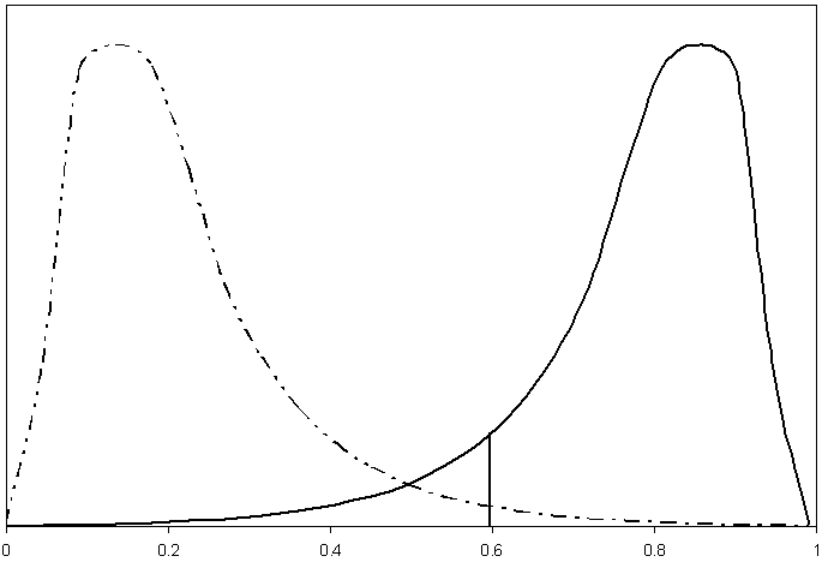

Fig. 5. Output neuron probability distributions under null and alternativehypothesis

More often than not, and for no apparent good reason, the threshold is set to exactly 0.5 in theoretical experiments. Square in the middle of neuron activation range is not necessarily the optimum place for an output neuron's activation threshold, and better generalisation performance might be achieved by shifting the threshold higher or lower. The probability of making a type I error is given by the area under the left curve to the right of the threshold, and is shown as the black area in figure 6. On the other hand, the probability of making a type II error is given by the area under the right curve to the left of the threshold, shown as the gray area in figure 6. It is obvious from the above that the higher the threshold is set, the smaller the chance becomes of making a type I error (false positive), but the bigger the chance of making a type II error (false negative). The probability of making a false positive decision is also called the *false alarm rate*: $FAR = P(false\ alarm|legitimate)$. In terms of table 2, the false alarm rate can be derived as

$$\begin{aligned} \text{FAR} &= \frac{\#\text{false alarms}}{\#\text{all alarms}} \\ &\geq \frac{\#\text{false alarms}}{\#\text{all alarms}} \frac{\#\text{all alarms}}{\#\text{all legals}}^{2} \\ &= \frac{\#\text{false alarms}}{\#\text{all legals}} \\ &= \frac{\text{FP}}{\text{FP}+\text{TN}} \end{aligned} \tag{28}$$

[2] Allowable since $1 \geq \frac{\#\text{all alarms}}{\#\text{all legals}}$ in most realistic cases [4].

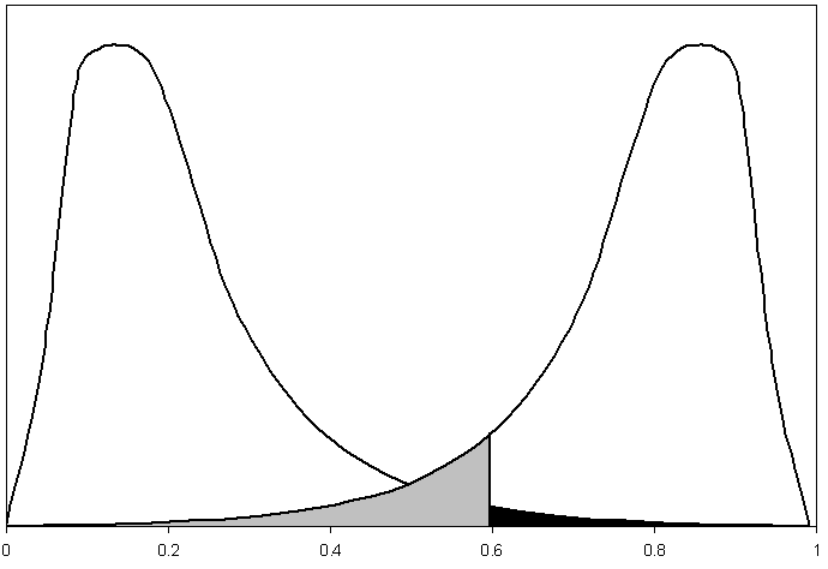

Fig. 6. Decision outcome error probability

Another quantity of interest is the *true positive rate* (or *hit rate*), which is the probability of making a true positive decision (i.e. $1 - P(\text{fraud not detected})$), and is given by

$$\begin{aligned} \text{TPR} &= 1 - \frac{\#\text{false negatives}}{\#\text{all alarms}} \\ &= \frac{\text{TP}}{\text{TP} + \text{FN}}. \end{aligned} \tag{29}$$

Together, these two quantities can give a good indication of a binary classifier's generalisation performance. Plotting these two quantities against each other for increasing thresholds from 0 to 1, with the false alarm rate on the x-axis and the true positive rate on the y-axis, yields a curve called the *Receiver Operating Characteristic* (ROC) curve (see figure 7). The ROC curve was first used in the Signal Detection Theory field as an indication of a radar operator's ability to distinguish between noise, friendly ships, and enemy ships when viewing blips on a radar screen. In the 1970's, the ROC curve found its way into the medical sciences where it was found useful for interpreting medical test results. Because of its success in describing binary classification performance, it was bound to find its way into the machine learning field too.

Originally, the classical concept of a detection threshold predicts a linear relationship between the FAR and TPR [13]. This can be shown by defining a quantity that captures the probability that the neuron activation will exceed a set threshold,

$$p = \frac{\text{TPR} - \text{FAR}}{1 - \text{FAR}}. \tag{30}$$

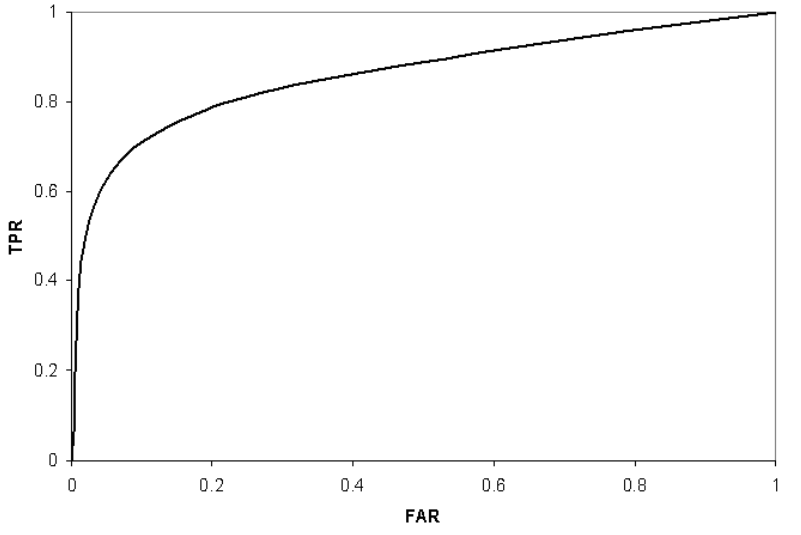

Fig. 7. A typical ROC curve

Rearranging (30) leads to the linear relationship between FAR and TPR predicted by the high threshold model:

$$\mathrm{TPR} = \mathrm{p} + (1 - \mathrm{p}) \cdot \mathrm{FAR} \tag{31}$$

The bowed-shaped ROC curve of figure 7 contradicts this linear relationship predicted by the high threshold model, and this is one of the reasons why the high threshold model was eventually abandoned and replaced with signal detection theory [13]. In signal detection theory, the sensory process has no threshold, but rather a continuous output based on random Gaussian noise with which the signal combines when it is present. The relationship between FAR and TPR can be expressed as z-scores of the unit, normal Gaussian probability distribution

$$z(\mathrm{TPR}) = \mu_s \cdot \frac{\sigma_\mathrm{n}}{\sigma_\mathrm{s}} + \frac{\sigma_\mathrm{n}}{\sigma_\mathrm{s}} \cdot \mathrm{z}(\mathrm{FAR}) \tag{32}$$

with μ_s the signal-plus-noise distribution, $\sigma_s$its standard deviation, and $\sigma_n$the standard deviation of the noise. Signal detection theory as applied to detection sensitivity is outside the scope of this chapter, but it is necessary to note a few points:

1. The ROC curve predicted by signal detection theory is anchored at the (0,0) and (1,1) points (figure 7);
2. When μ_s is greater than zero, the ROC curve takes on a bowed-shape;
3. When μ_s is zero the ROC curve is a straight diagonal line connecting (0,0) and (1,1).

For the purpose of measuring performance in this chapter we are only interested in the total area under the ROC curve, called the area-under-curve (AUC). The area under

the ROC curve gives an indication of a classifier's ability to generalise and the bigger this AUC value, the better the classifier. A perfect classifier will have a ROC curve that resembles a right angle with an enclosed area of unity.

The results of experiments conducted in section 6 will include both the MSE and ROC curve area as performance measures. We now have two methods for fraud detection, and a performance measure applicable to our specific problem. The final step before our methodologies can be put to the test is processing the data set to make it compatible with our learning algorithms. Feature selection and preprocessing is discussed in the following section.

5 Data Sets, Feature Selection and Preprocessing

This section deals with the tedious task of selecting appropriate features from a real world data set and applying proper preprocessing techniques to them prior to their presentation to a network for classification. Exact details of the data set used here cannot be made public due to its intrinsic sensitivity. This is a typical problem hampering transaction fraud research: no common, publicly available transaction database seems to exist with which one can make a comparison between techniques described in fraud detection literature. The best we can do is to at least, in the context of this chapter, decide which of our two methodologies performs better in an attempt to gauge whether further research in such a direction is worthwile or not.

In the context of this chapter a feature represents a unit of data chosen from the list of available data columns present in a data set, and will ultimately form one or more dimensions of a network input vector after the feature has been sufficiently preprocessed. Preprocessing is the task of taking raw features and converting them into values that are suitable for network presentation and classification. Each row in a data set translates to a network input vector when feature selection and preprocessing steps are applied to it. The different features present in a data set can normally be categorised into four classes according to the type and degree of information they contain [22]. The remainder of this section deals with the selection of features from a data set for fraud detection purposes, the different categories they can be divided into according to their type, and the preprocessing techniques used to convert the features in each category to numerical values that can be used as inputs to neural networks.

5.1 Feature Selection

Selecting a salient set of features from a data set and applying the correct preprocessing techniques to them more often than not means the difference between success and failure when building a neural network for classifying real world data. Apart from obvious transaction features such as the amount, what other data features should be selected ? Available transaction information differs from card issuer to card issuer, and details on this type of information is seen as proprietary and are therefore never published. An even bigger problem inherent in data sets used for fraud detection is the high number of symbolic fields present in the transactional data, while neural networks and other classifier algorithms can only deal with numeric data. The reason for this is that authorisation message exchange between point-of-sale terminals and card management

platforms were simply not designed with fraud detection in mind. We might be tempted to simply ignore symbolic fields and only present the numerical data present in the data set to the classifier, but the information contained in symbolic fields can potentially be just as important in correctly classifying credit card transactions as the information contained in the numeric fields. Take for example the transaction date; most conventional fraud detection and expert systems will ignore this field because it is seen as unimportant information [4]. However, a fraud system that attempts to model a time series will obviously find this information very useful. Another reason for using the transaction date might be to model the changes in human shopping behaviour as time progresses. Conventional fraud detection systems seem to have difficulty in coping with changing shopping profiles, especially during holiday seasons.

Although the information contained in authorisation messages might be seen as limited when attempting fraud classification, some statistical values can be calculated to serve as additional features in the input vector. An example of such a statistical value used in conventional fraud classifiers is the transaction velocity. In the context of fraud detection, the velocity value of an account at any given point in time is calculated by counting the number of transactions done on the account during a pre-specified timeframe. The rationale behind this is that if a card is, for example, used to buy five different flight tickets or do ten purchases at different jewellery stores in a short period of time, the activity should be regarded as highly suspicious. Different velocities can be calculated by including only transactions done at certain types of merchants in one velocity calculation, and including all transactions in another. Merchants can be grouped into industry for the above mentioned velocity calculation by using the standard industry code (SIC) present in every authorisation message. The SIC itself, although symbolic, can also be included in the input vector as a feature. The use of time-based velocity statistics in the conventional fraud detection systems in use today gives a clue to the importance of time when dealing with credit card fraud. After all, the introduction of statistical velocity values ultimately introduces a dynamic component into an otherwise static classification methodology. Demographic information about the card holder can itself also prove useful as one or more dimensions of a feature vector.

Table 3 lists some of the raw features used in our fraud detector and also which variable class they fall into. Each feature is a variable with distinct content and data ranges, and it is this information that is central to classifying a feature into one of the four major variable classes, which in turn will tell us how to approach the feature during pre-processing.

The transaction date and amount are obvious choices, with the date on which the transaction was made important for giving the classifier a chance to overcome behavioural changes due to seasonal changes. The billing amount is included to provide some stabilisation around the amount feature, since the billing amount will always be in the same currency for an account as opposed to the transaction amount where the currency depends on the currency of the merchant where the transaction was made.

The merchant identifier only serves to tell the classifier whether consecutive transactions on an account are done in the same country or not. If two consecutive transactions are done in different countries within, say, 10 minutes from each other, then something is wrong and it might point to a skimmed card being used. The standard industry code

Table 3. A subset of the features used to train our classifiers

Data item	Variable type
Transaction date	Interval
Transaction amount	Ratio
Billing amount	Ratio
Merchant identifier	Nominal
Merchant standard industry code	Nominal
Card holder birth date (age)	Interval
Card holder membership tenure	Interval
Card holder country	Nominal
Card renewal date	Interval
Last cycle credits	Ratio
Last cycle debits	Ratio
Last cycle outstanding amount	Ratio
Velocity calculation 1	Interval
Velocity calculation 2	Interval

is used to highlight when the same type of goods or services are purchased repeatedly. Some type of merchants are higher fraud risks than others, and the SIC also helps to identify this. Demographic features like customer age, membership tenure and country of residence can be used to train a classifier to identify when a customer falls into a high risk segment, while the card expiry or renewal date is useful to note whether transactions on an account are done with a new card. The last five features in table 3 are statistic in nature. The cycle information shows the spending and payment pattern during the last billing cycle, and a classifier can learn to identify devations from the norm. The two velocity values were already discussed earlier.

The following two subsections deal with identifying correct preprocessing methods for each feature once the selection process is finished, based on what type of variable class they belong to.

5.2 Nominal and Ordinal Variables

Variables are considered nominal when they cannot be measured in a quantative way and only contain information to whether they form part of a pre-specified category or not. The only mathematical test that can be done on the nominal variables of a specific category is whether they are equal or not. One nominal variable cannot be considered greater or smaller than another nominal variable of the same type.

Nominal variables are usually presented to a neural network by using as many neurons as the number of distinct values within the category to which the variable belong. One of the neurons is then activated to present a specific value in the category, while all the remaining neurons remain deactivated. This is called *one-of-n* or *one-hot* encoding. This approach works well when the range of values the variable can take on is small, like for instance gender that can only take on one of two values (male or female). When the number of values increases though, this method runs into some difficulties:

1. When the number of distinct values are large, the vectors obtained by applying *one-of-n* encoding are so similar that a network might have difficulty in learning the difference between different categories of the same nominal variable [22].
2. Most nominal variables in real world data can have an extremely large number of categories, like for instance postal code, country code, standard industry code, etc. Representing such variables using *one-of-n* encoding is computationally infeasible most of the time.

An alternative approach that seems to work quite well is to compute a ranking score for each category value a nominal variable of a specified type can have. The ranking score can then be further pre-processed and used as input to a neural network. The ranking scores for a particular nominal variable are computed by simply counting the number of occurrences of each category within the data set and sorting the resulting counts. The category with the least number of occurrences will then be ranked first, while the category that occurs the most will be ranked last.

Variables that have a true implicit order between categories, but whose physical categorical values have no meaning other than establishing order, are called ordinal variables. Since computing ranking scores for nominal variables almost seems to turn them into ordinal variables, one might be tempted to use the ranking score directly as a dimension of the input vector by encoding the value into one neuron, which is allowable for ordinal variables. The problem with this is that, by encoding the ranking score into a single neuron, it is implied that an order relationship between distinctive categories are present where there is in fact none. Therefore, the ranking scores computed for experiments in this chapter are first converted to binary numbers and then encoded into a fixed number of neurons for each variable. The number of neurons required for binary encoding can be computed using equation (33), where c denotes the number of possible categories the nominal variable can take on.

$$n = \left\lceil \frac{\log c}{\log 2} \right\rceil \tag{33}$$

5.3 Interval and Ratio Variables

Interval variables are numeric in nature, have definite values, and have order relationships between different values of the same variable. Interval variables are usually presented to a neural network using one neuron. Features such as transaction velocity and date - when broken up into days, months and years - are interval variables. Ratio variables are the same as interval variables, except that a value of zero in a ratio scale is a true zero, while it is arbitrary on an interval scale. The transaction amount is an example of a ratio variable. In order to present interval or ratio variables to a neural network, they must first be normalised to prevent them from saturating the input neurons. In other words, they must be scaled to adhere with the activation limits of the input neurons of a particular network type. The simplest way to normalise interval or ratio variables is to calculate the minimum (X_{min}) and maximum (X_{max}) values that a variable x can take on in both the test and training set, and then use equation (34) to scale the value to

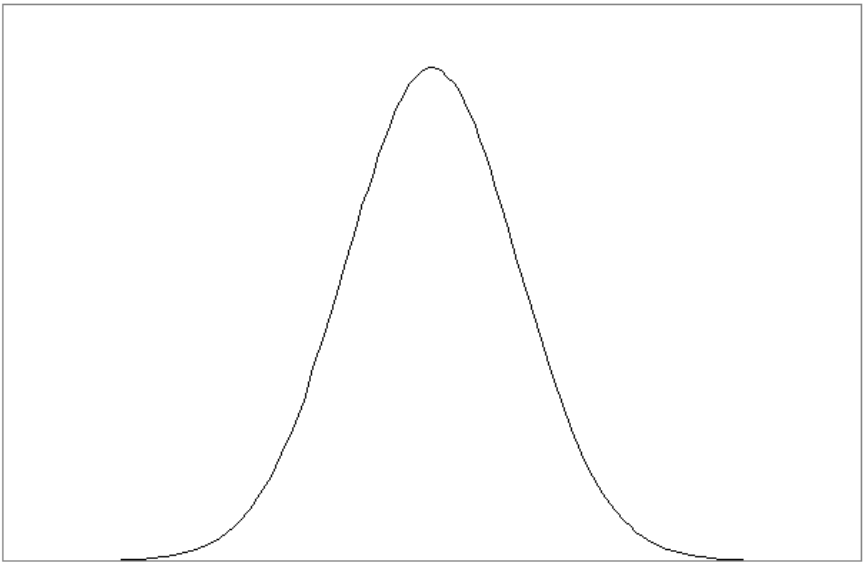

Fig. 8. A normal distribution with mean 10 and standard deviation 2

fall within the network input range, [I_{min},I_{max}]. The network input range is arbitrarily chosen, and is set to [-1,1] for all the experiments in this chapter.

$$y = \left(\frac{x - X_{min}}{X_{max} - X_{min}} \right) (I_{max} - I_{min}) + I_{min} \tag{34}$$

Equation (35) shows an alternative approach for normalising a value. This statistical approach can be used when the set of values of a feature has the tendency to cluster around some particular value and therefore exhibits an approximately normal distribution [25].

$$y = \frac{x - \mu}{\sigma} \tag{35}$$

Equation (35) states that a value x can be normalised by subtracting its *mean* μ and dividing the result by its *standard deviation* σ. The main reason why equation (35) is usually preferred over equation (34), is because it removes all effects that offset and measurement scale can have on a feature's values [22]. Equations (36) and (37) below show the well known formulae used to compute the mean and standard deviation of a feature.

$$\mu = \frac{1}{N} \sum_{j=1}^{N} x_j \tag{36}$$

$$\sigma = \sqrt{\frac{1}{N} \sum_{j=1}^{N} (x_j - \mu)^2} \tag{37}$$

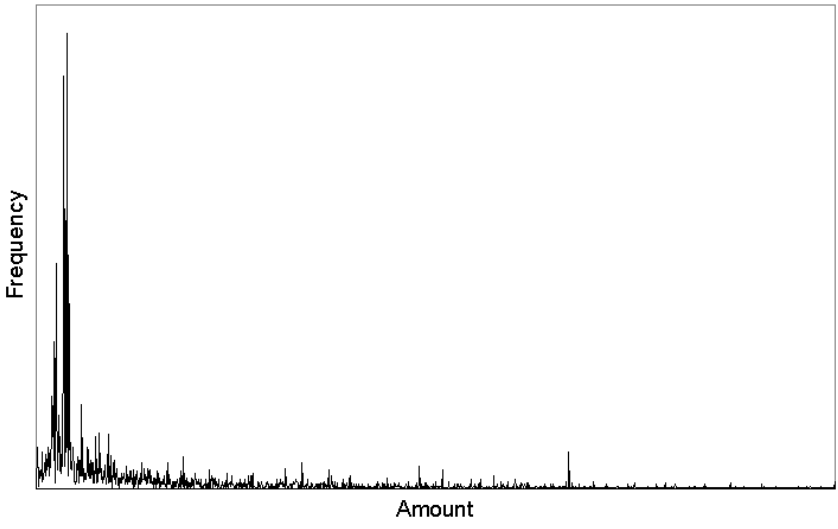

Fig. 9. A histogram of the *transaction amount* feature prior to preprocessing

Figure 8 shows an illustration of what a normal distribution might look like. The only downside to using equations (34) or (35) for normalisation purposes is the fact that they do not deal particularly well with outliers in a data set. Outliers will cause the bigger part of the data to be scaled into a relatively small section of the network input range, causing these values to have much less of an impact on training and generalisation then they should. Figure 9 shows a histogram of the *transaction amount* feature. It is clear from the histogram that the information content of the variable is completely distorted and does not resemble anything even close to a normal distribution. Most neural networks will have a hard time learning anything from a feature such as the one depicted here, unless it is first transformed using a compression transformation to stabilise its variance. In most real life data sets the variance of each observation does not remain constant, and it is this unevenness in the data that we try to remedy through the use of a compression transformation. The logarithm is one of the more commonly used transformations and also the one used in the preprocessing steps for the experiments in this chapter. Figure 10 shows the histogram of the transaction amount feature after being passed through the logarithmic compression transformation.

Equation (38) shows an improved transform first suggested by J. Tukey (cited in [1]), and figure 11 shows the effect this transformation has on the distribution of the transaction amount feature.

$$y = \arcsin\left(\sqrt{\frac{x}{n+1}}\right) + \arcsin\left(\sqrt{\frac{x+1}{n+1}}\right) \tag{38}$$

The transform in Equation (38) did not seem to have much of an effect on the training and generalisation performance in the experiments of this chapter, and was therefore not used in the final version of the preprocessing algorithm. After transformation and normalisation, the transaction amount feature still does not quite resemble a normal

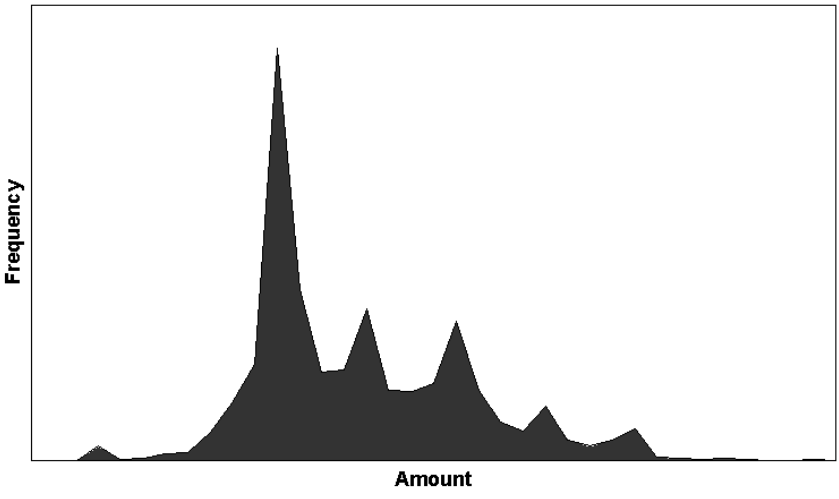

Fig. 10. A histogram of the *transaction amount* feature after applying a simple logarithmic compression transformation

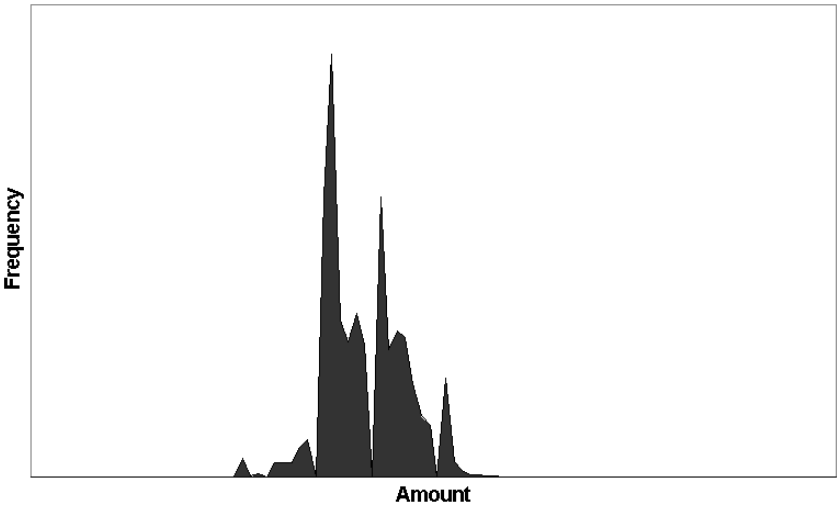

Fig. 11. A histogram of the *transaction amount* feature after applying the transform in equation (38)

distribution, and from Figure 11 it is clear that its distribution is plagued by a fair amount of skewness and kurtosis; the skewness characterises the degree of asymmetry of a distribution around its mean, while the kurtosis measures the peakedness or flatness of a distribution [25]. Nevertheless, the methods discussed in this section had the desired effect on features and significantly improved both the training and generalisation performance of the classification algorithms used in the experiments in this

chapter. The next section puts the theory presented in the past four sections to the test in a series of experiments on our real world data set.

6 Detecting Credit Card Fraud

In this section, the machine learning methodologies discussed in section 2 and section 3 are put to the test on a real world data set containing a mixture of legitimate and fraudulent credit card transactions. In the first experiment, we attempt to detect fraud using an SVM. The data set used for this experiment contains a total of 30876 transactions evened out over 12 months. The second experiment is done with the same data, but with the transactions time-ordered to test whether LSTM can manage to learn the time series inherent in the data set. The resulting transaction sequences are of variable length; the sequence length distributions of the training and test data sets are shown in figure 12.

In each of the experiment subsections, the set up is explained and the results presented, after which a discussion follows. The implementations of the machine learning algorithms used in the experiments are also discussed at the start of each subsection. The performance measurement tool *PERF Release 5.10* [7] was used in each case to calculate the various performance metrics reported.

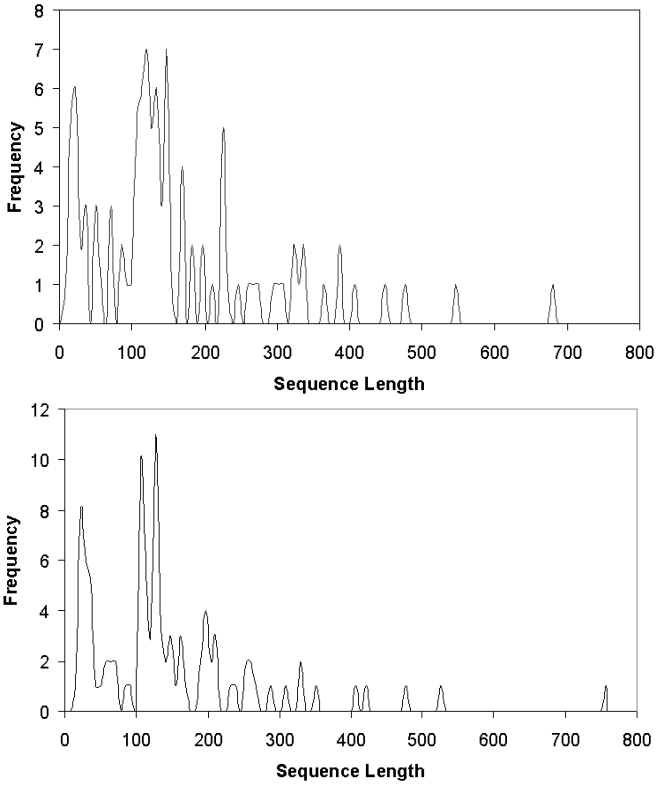

Fig. 12. Sequence length distributions of the training set (top) and test set (bottom)

6.1 Experiment I: Fraud Detection with SVM

Experimental Set Up. The original data set was manipulated by decreasing the number of legitimate transactions to exhibit a class distribution of 99 to 1 (99% legitimate vs. 1% fraud). This was done to reduce the overall size of the data set, making it more manageable for use in our experiments. This is also the distribution used in other payment card fraud detection literature [4]. The distribution of the original data set was of course even more skewed, containing less than 0.1% fraud. The result of the manipulation was a data set consisting of 30876 transactions which was in turn split into two equal parts, one for training and another for testing, by randomly selecting sequences of same card transactions. By same card transaction sequences we mean a time ordered sequence of transactions done with a particular credit card. Transactions of a paticular credit card account can belong to either the training or the test set, but not both. The transactions contained in these two sets are equally spaced out over the 12 months of an arbitrary year, and were pre-processed using the techniques described in the previous section to finally obtain a 41-dimensional input (feature) vector with a corresponding class label for each transaction.

Training and generalisation test cycles are normally run a total of 30 times for neural network based algorithms to ensure that a statistically meaningful sample is obtained and to ensure that poor performance due to local minima is ruled out. For SVM on the other hand, since its performance depends on what values the kernel parameters and cost function are set to (and since SVMs do not exhibit the randomness inherent in neural network learning), time was rather spend estimating the best values for these parameters than repeating the same experiment 30 times.

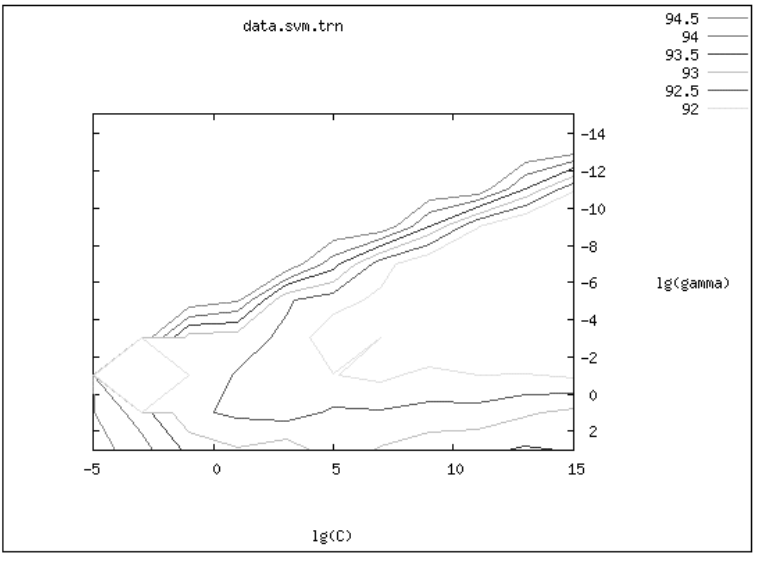

Fig. 13. Grid-search results for SVM parameter selection

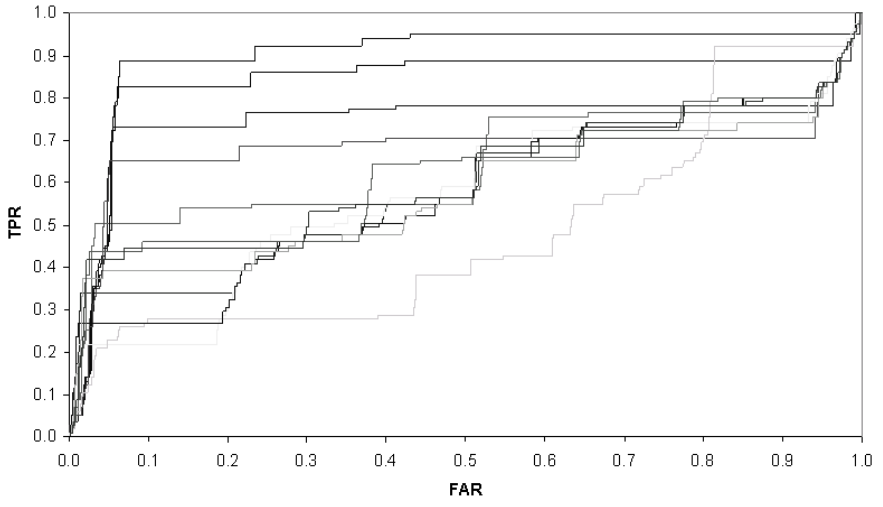

Fig. 14. ROC curves for various SVM kernels, parameters, and cost

The SVM has been around for a while and many tried and tested off-the-shelf packages are freely available with which SVM researchers can experiment. In this particular case, Chang and Lin's popular *libSVM* [8] implementation was used which includes a wide variety of tools to make dealing with SVM classification much simpler for the novice user.

Parameters and Training. To a great extent the performance of an SVM depends on what kernel is used and what the values of the kernel parameters and cost are set to. Since SVM training guarantees to always find the maximum margin achievable for a given set of parameters, it is fruitless to conduct multiple trial runs with the same parameters and time should rather be spent finding the most appropriate kernel and its parameters. Unfortunately, no real satisfying heuristic exists with which to compute a kernel's parameters and in most cases the best one can do is to guess. The search for the best parameters, however, can be done in a structured manner. It was already mentioned in section 2.2 that a combination of cross-validation and grid-search is a popular method for obtaining a best guess estimate of the cost parameter (C) and the kernel parameter (γ) when dealing with radial basis function (RBF) kernels. Cross-validation can also help to prevent over fitting of the training set [15]. The cross-validation and grid-search methods were utilised here to obtain a best guess estimate of these parameters.

Cross-validation involves splitting the training set into two parts. Training is then done on one part and testing on the other. In v-fold cross-validation, the training set is divided into v parts and the classifier sequentially trained on v-1 of the subsets, and then tested on the remaining subset. This way, each subset is tested once and each training instance is predicted once, and the cross-validation accuracy is therefore the percentage of training examples correctly classified. Grid-search is a straightforward and simple technique. To estimate the best parameters for a given classification problem and kernel, a growing sequence of parameter values are used for training and the ones giving the

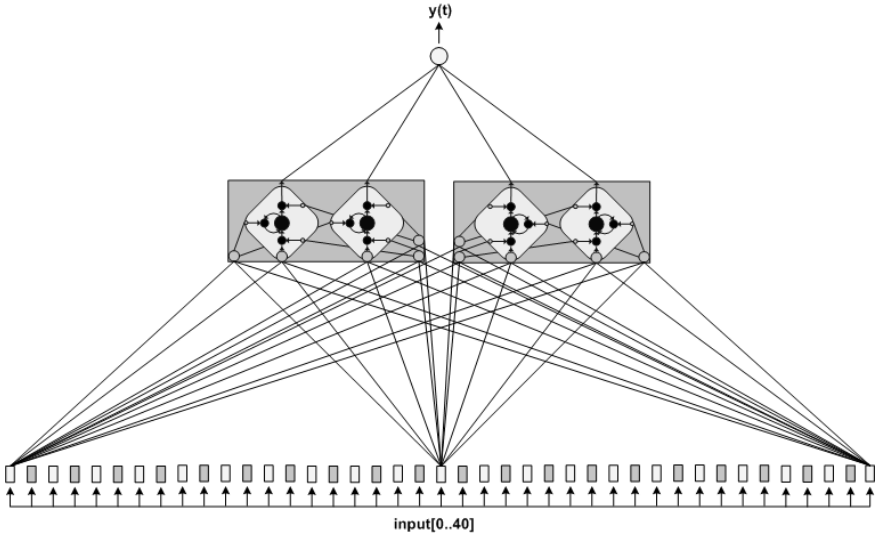

Fig. 15. LSTM network topology for fraud detection

best cross-validation accuracy is picked. In case of the RBF kernel, the parameter values consist of (C,γ) pairs. Cross-validation and grid-search were done using the parameter selection tool *grid.py* that comes packaged with *libSVM*.

Figure 13 shows the output of *grid.py* as applied to fraud detection during one of the parameter selection runs. Since the class distribution of the training set is skewed and biased towards the legitimate transaction class, the cost of misclassification amongst the two classes are different. On the one hand, one might want to cost the misclassification of fraudulent transactions more because

1. not detecting them directly translates into monetary loss, and
2. fraud transactions in the data set are sparsely represented.

On the other hand, one might also want to cost the misclassification of a legitimate transaction more since customer dissatisfaction can be more harmful than fraud in some cases. In cases where appropriate cost parameters are not introduced for skewed data sets, underfitting of the data is likely to occur. In such cases, the classifier will assign the legal transaction label to all transactions, making it impossible to detect any fraud. During the parameter grid-search for this experiment, different ratios of cost for the two classes were tried and the best cost ratio was found to be 1:10, with mistakes on the fraud class being penalised 10 times more than mistakes on the legitimate class. The different kernel types tested included linear, polynomial, RBF, and sigmoid kernels.

Results. Figure 14 shows the ROC curves of various trial runs with different kernels, kernel parameters and cost ratios. The best generalisation performance on the test set was achieved by using an RBF kernel with $\gamma = 11$, cost $C = 0.05$ and cost ratio 1:10

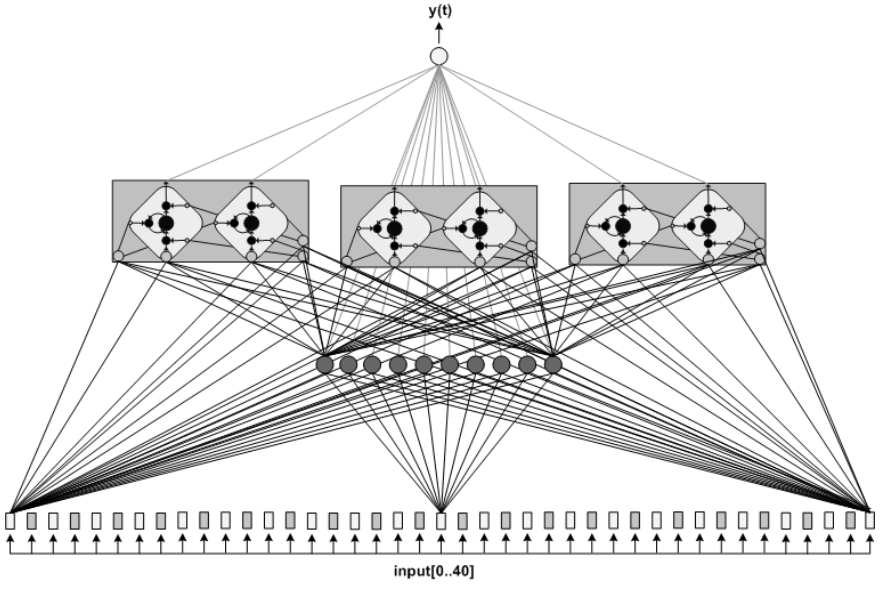

Fig. 16. A hybrid LSTM with one hidden layer

as already mentioned. The AUC for this curve was found to be 0.89322. The SVM achieved a classification rate of roughly 213 transaction/second.[3]

6.2 Experiment II: Time Series Modelling with LSTM

A bespoke C/C++ implementation of LSTM (called *LSTMFraud*) was used in this experiment. The implementation is executable in both the WIN32 and Linux environments and was benchmarked against the Extended Reber Grammar problem [11] to ensure that it at least matches the performance of the original implementations of Hochreiter and Gers used in the initial LSTM experiments [14][11]. The algorithm was implemented and customised rather than using one of the downloadable implementations because we felt that a much better understanding of the algorithm can be gained by actually implementing it from scratch. All of the important parameters are customisable in *LSTMFraud*, for instance the number of memory blocks, memory cells per block, hidden neurons, output neurons, the learning rate, rate decay, and also whether input-output shortcuts and peephole connections should be used. Various test runs were executed in order to find the best combination of these parameters prior to executing the 30 trial runs of the experiment.

Network Topology, Parameters, and Training. The use of LSTM by nature results in a near fully connected network. Depending on implementation preference and parameter

[3] This was calculated in a WIN32 environment, and according to [18] the Windows version of libSVM sometimes suffers inexplicable delays during training. This might also be true for classification.

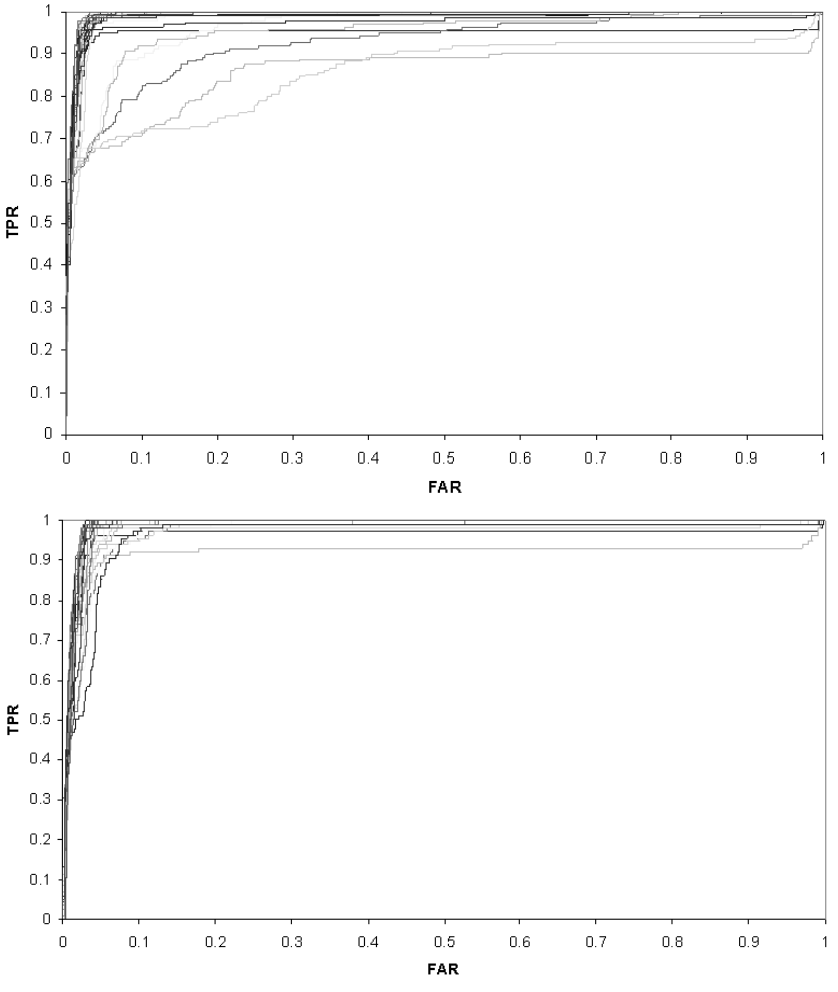

Fig. 17. 30 ROC curves measuring LSTM fraud detection performance on the training set (top) and test set (bottom)

settings, almost every single output signal flows into the inputs of every single neuron entity present in the network. Figure 15 shows the LSTM topology used for this experiment. Two memory blocks with two cells each were used. Not all connections can be shown due to reasons of intelligibility; for instance figure 15 does not show feedback connections between cell outputs and memory block and cell inputs. Each trial run consisted of feeding pre-processed randomly selected transaction sequences into the network for 100 epochs, where an epoch is the size of the training set (in this case the training set contaned 16263 transactions). Although LSTM does not seem to suffer as much from problems with overfitting as normal feedforward neural networks (FFNN) do, 100 epochs were chosen because this resulted in the best generalisation performance during initial experimentation. The network was not manually reset between sequences

or epochs, and no learning rate decay, peephole connections or input-output shortcuts were used. The learning rate for this expirement was set to 0.3. During training, whenever a misclassification occurred, training on the transaction sequence was immediately restarted. This was repeated until all transactions of a sequence were correctly classified, after which a new sequence was then randomly selected from the training set for the next iteration. At the end of each training cycle, the fraud probability for each transaction in both the training and test set was recorded in order to draw the ROC curves and calculate the AUC and MSE. During generalisation testing, the network is reset after each transaction sequence. The transaction sequences used in both training and testing were of variable length.

A hybrid network with one feedforward hidden layer was also tested, but no real improvement in generalisation performance was noted (see figure 16). The motivation for its initial use was that, in seperate experiments, FFNN performed well on the data set too, and that a combined FFNN-LSTM approach might lift the generalisation performance somewhat. The problem with this approach is that static FFNN neurons learn much quicker than their LSTM counterparts, and their weights will therefore start overfitting the training data long before the LSTM memory cells are fully trained. One possible solution to this problem is to have separate learning rates for the FFNN and LSTM parts of the hybrid network; however, it remains difficult to obtain the correct ratio between these two rates. The training method employed here, i.e. stopping the presentation of a sequence once a misclassification occurs and then re-presenting it, is not really applicable to FFNN training either, where it is better to present as much of the data set as possible during each epoch.

Results. Figure 17 shows the generalisation ROC curves of the 30 trials with LSTM. Their deviation from a straight diagonal between (0,0) and (1,1) and consequent tendency to bow towards the (0,1) corner of the graph shows that the algorithm actually learned something during training and that it performs far better than a constant "not fraud" diagnosis would, despite the overwhelming bias towards the legitimate transaction class in the test data set. The spread of the ROC curves shows the level of randomness inherent in neural network training where initial weight values are randomised leading to a different gradient search start position in weight space each time the network is re-initialised prior to training. In addition, the transaction sequences are also randomly picked from the training set for each iteration through the network. Table 4 shows the maximum, minimum, and average AUC values recorded during the training and testing stages of the LSTM time series experiment, including their corresponding MSE values, and also the 95% AUC confidence interval calculated using the results of all 30 trials. The maximum AUC recorded on the test set during the 30 trials is 0.99216, and the minimum 0.91903. Training time for LSTM is slower than that of SVM, but classification is fairly quick with a recorded classification rate of approximately 2690 transactions/second[4]. Table 5 lists all results for LSTM over the 30 training and 30 test trials, in the order in which they were recorded.

[4] Classification rates reported here are indicative. Classification rates were calculated on an Intel Celeron D 2.66 Ghz processor, and excluded pre-processing of the feature vectors.

Table 4. Summary of training and test results for LSTM

Rank #	Training Set		Test Set	
	AUC	MSE	AUC	MSE
Maximum	0.99524	0.12660	0.99216	0.12886
Minimum	0.85868	0.12942	0.91903	0.12938
Average	0.97571	0.12873	0.98221	0.13128
95% Confidence Interval	0.96326 - 0.98817		0.97716 - 0.98727	

Table 5. Full list of training and test results for LSTM

Trial #	Training Set		Test Set	
	AUC	MSE	AUC	MSE
1	0.99264	0.12684	0.99091	0.12435
2	0.99145	0.13085	0.98844	0.14216
3	0.99085	0.12265	0.98929	0.12163
4	0.99333	0.13107	0.98890	0.12571
5	0.94763	0.12837	0.96308	0.12674
6	0.98994	0.13235	0.98889	0.13450
7	0.92630	0.13210	0.97610	0.12552
8	0.98680	0.12702	0.98313	0.12830
9	0.99202	0.11228	0.99115	0.10950
10	0.98512	0.12873	0.98993	0.13407
11	0.96602	0.13294	0.98687	0.12673
12	0.99041	0.12514	0.97613	0.13880
13	0.99367	0.12749	0.98877	0.13715
14	0.95212	0.13233	0.98596	0.12686
15	0.98996	0.12883	0.98380	0.14431
16	0.98638	0.13916	0.96679	0.13910
17	0.99183	0.13011	0.99216	0.12886
18	0.99266	0.13376	0.98144	0.13506
19	0.85868	0.12942	0.91903	0.12938
20	0.86001	0.12564	0.97470	0.13521
21	0.99141	0.12703	0.98910	0.12236
22	0.99397	0.12969	0.99145	0.12802
23	0.99524	0.12660	0.98191	0.13336
24	0.98505	0.12755	0.98854	0.13442
25	0.98913	0.12851	0.99060	0.13729
26	0.99359	0.12725	0.98934	0.12809
27	0.97382	0.12941	0.96552	0.13253
28	0.98631	0.12788	0.98768	0.13090
29	0.99249	0.13030	0.98966	0.13696
30	0.99256	0.13066	0.98717	0.14065

6.3 Discussion

Both algorithms tested here achieved remarkably good separation between the legitimate and fraud classes in the test data set. Figure 18 shows the ROC curves of the best performing SVM and LSTM instances and Table 6 lists the AUC values of these ROC curves, along with the classification rates for each algorithm. As postulated, LSTM on time-ordered data outperformed SVM - a remarkable feat if one takes into account that SVM performed quite well too. It is quite possible that the optimum SVM kernel and kernel parameters were not used for this particular data set; searching for these parameters is a time consuming task and we did the best we could with cross-validation and grid-search in the time available. Nevertheless, SVM still achieved a respectable AUC of 0.89322. Training times were shorter with SVM, but classification was quickest with LSTM at 2690 transactions/second, compared to 213 transactions/second for SVM. It is interesting to note that the average AUC computed for LSTM on the test set was actually higher than on the training set. Having a closer look at the full list of results (table 5) one sees that in most cases performance on the training set was either better than on the test set, or more or less the same as on the test set; for trials 7, 19 and 20, however, performance on the training set were worse by such a large margin that it had an excessively negative effect on the average performance of the classifier during training. When

Table 6. Comparison of SVM and LSTM

Method	Max AUC	Average AUC	Classification Rate (transactions/sec)
SVM	0.89322	n/a	213
LSTM	0.99216	0.98221	2690

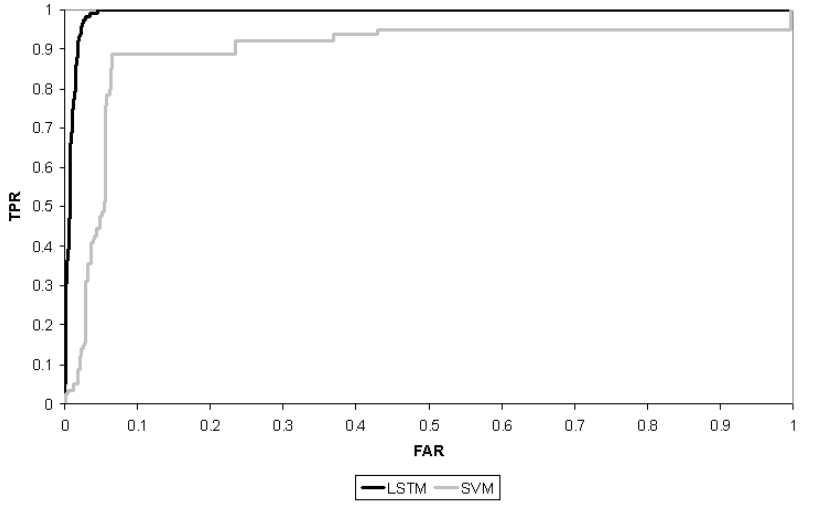

Fig. 18. ROC curves of best performing LSTM and SVM classifiers

one has another look at figure 12 though, one notices that the average sequence length of the training patterns is longer than the average length of the test patterns. In fact, the average sequence length in the training set was calculated as 160.85, as opposed to an average length of 135.77 for the test set. Longer sequences are more difficult to classify correctly and we can postulate that the larger average sequence length in the training set is to blame for the weaker generalisation performance during training. Significantly better performance on the test set in some instances are certainly unusual, but we will not investigate it any further here since we feel it will distract from the main objective of this chapter.

7 Conclusions and Future Research

7.1 Conclusion

In this chapter, we set out to show what can be gained in generalisation performance by modelling the time series inherent in sequences of credit card transactions as opposed to dealing with individual transactions. To support these claims, we proposed two fraud transaction modelling methodologies for analytical comparison: support vector machines and long short-term memory recurrent neural networks. The results of the experiments conducted in section 6 largely confirmed our initial hypothesis, and LSTM proved to be a highly successful method for modelling time series. Our claim was further strengthened by the fact that the fraud detection systems in use today, use statistics such as the transaction velocity [23] and distance mapping techniques between successive transactions to aid them in discriminating legitimate from fraudulent transactions, which in essence introduce a concept of time and dynamics into otherwise static methodologies anyway. What makes the success of LSTM more remarkable is that the time series modelled here are of variable length and different for each card member. LSTM therefore modelled a set of completely diverse time series and still managed to outperform a very popular machine learning method. We believe that LSTM can be used as a remedy to the bad performance due to changing shopping behaviour in fraud detection systems.

We are, however, careful to put any claims forward based on the results because the technique used to estimate the kernels and parameters for SVM training is definitely not infallible. It is stated in [15] that grid-search is sometimes not good enough, and techniques such as feature selection need to be applied. Here, we have already done feature selection and preprocessing prior to applying grid-search; so, the only other logical explanation for SVM's weaker performance must be based on the kernel used. SVM's weaker performance might also be attributed to outliers in the data set that were not removed prior to training; the data set used here definitely contained some noise. One has to, however, be careful of making statements regarding outliers and their effect (or lack thereof) on generalisation performance.

Comparing the performance of the two methodologies used here to that of other documented techniques is difficult, largely because not all of the fraud detection literature available uses AUC as a performance measure, but mostly because a common data set is not used, which makes any comparative findings indefinite. If one is forced to do a comparison though, Brause et al. [4] reported a TPR of 0.37 at a FAR of 0.06 on their

fraud sequence diagnostic problem, while LSTM achieved an average TPR of 0.980283 for the same FAR during the 30 trials on our specific data set. Maes et al. [21] reported a TPR of 0.68 at a FAR of 0.1, and 0.74 at a FAR of 0.15, while our LSTM network achieved on average 0.98898 at a 0.1 FAR and 0.99449 at a 0.15 FAR over the 30 trial runs.

Finally, to put all these results into perspective, we would again like to state that our data set, because it is real, is not only noisy, but also has an extremely skewed class distribution. In the light of this, both methods actually performed well, a fact we can largely contribute to proper feature selection and preprocessing.

7.2 Future Research

Here we applied LSTM to a set of dissimilar time series of variable length. This is a gross over complication of the fraud detection problem and future research effort can rather be spent towards using LSTM to model singular time series. This will enable us to form a better insight into how well LSTM copes with subtle variations in transactional time series. Note that the LSTM network topology used here was quite small, containing only two memory blocks with two cells each (it seems like Occam's Razor played a role again). This leads to a network with fewer weights and opens up the possibility of actually storing these weights on the chip of the next generation payment cards. This will enable point-of-sale terminals to do initial fraud detection using weight values stored on the chip, and only forward transactions with a high probability of fraud to the host system for further fraud detection processing. This approach will remove a lot of the burden from the host systems which will make true online fraud detection a possibility - a scenario in which fraud detection will equal fraud prevention. It will also be worthwhile investigating unsupervised learning methods as applied to fraud detection. In many cases where fraud detection systems are implemented in banks or countries that are new to fraud detection, one often finds that they have not kept or even identified any data as fraudulent, making supervised training impossible. Other research possibilities include rule extraction for validation and verification; the fusion of LSTM with other fraud detection methods like hidden-markov models or biometrics; and feature ranking using Neyman-Pearson hypothesis testing.

References

1. Acton, F.S.: Analysis of straight-line data. Dover Publications (1959) (1994)
2. Bolton, R.J., Hand, D.J.: Statistical Fraud Detection: A Review. Statistical Science 173, 235–255 (2002)
3. Boser, B.E., Guyon, I.M., Vapnik, V.N.: A training algorithm for optimal margin classifiers. In: Haussler, D. (ed.) Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory, pp. 144–152 (1992)
4. Brause, R., Langsdorf, T., Hepp, M.: Credit Card Fraud Detection by Adaptive Neural Data Mining (1999), http://Citeseer.ist.edu/brause99credit.html
5. Burges, C.J.C.: A tutorial on support vector machines for pattern recognition. Data Mining and Knowledge Discovery 2, 121–167 (1998)
6. Card Fraud: The Facts, The definitive guide on plastic card fraud and measures to prevent it. APACS (2005), http://www.apacs.org.uk

7. Caruana, R.: The PERF Performance Evaluation Code (2004), http://kodiak.cs.cornell.edu/kddcup/software.html
8. Chang, C., Lin, C.: LIBSVM: a library for support vector machines (2001), http://www.csie.ntu.edu.tw/~cjlin/libsvm
9. Cortes, C., Vapnik, V.: Support-Vector networks. Machine Learning 203, 273–297 (1995)
10. Elman, J.L.: Finding structure in time. Cognitive Science Journal 142, 179–211 (1990)
11. Gers, F.A., Schmidhuber, J., Cummins, F.: Learning to Forget: Continual Prediction with LSTM. Neural Computation 12, 2451–2471 (2000)
12. Gers, F.A., Schraudolph, N.N., Schmidhuber, J.: Learning Precise Timing with LSTM Recurrent Networks. Journal of Machine Learning Research 3, 115–143 (2002)
13. Harvey Jr., L.O.: Detection Sensitivity and Response Bias. Psychology of Perception. Psychology 4165, Department of Psychology, University of Colorado (2003)
14. Hochreiter, S., Schmidhuber, J.: Long Short-Term Memory. Neural Computation 9(8), 1735–1780 (1997)
15. Hsu, C., Chang, C., Lin, C.: A Practical Guide to Support Vector Classification. Technical Report, Department of Computer Science and Information Engineering, National Taiwan University (2003)
16. Jordan, M.I.: Attractor dynamics and parallelism in a connectionist sequential machine. In: Proceedings of the Eighth Annual Conference of the Cognitive Science Society, pp. 531–546 (1986)
17. Keerthi, S.S., Lin, C.: Asymptotic Behaviors of Support Vector Machines with Gaussian Kernel. Neural Computation 15, 1667–1689 (2003)
18. Kroon, S., Omlin, C.W.: Getting to grips with Support Vector Machines: Application. South African Statistical Journal 282, 93–114 (2003)
19. Kroon, S., Omlin, C.W.: Getting to grips with Support Vector Machines: Theory. South African Statistical Journal 282, 159–172 (2004)
20. Lee, Y., Lin, Y., Wahba, G.: Multicategory Support Vector Machines. Technical Report TR1040, Department of Statistics, University of Wisconsin (2001)
21. Maes, S., Tuyls, K., Vanschoenwinkel, B., Manderick, B.: Credit Card Fraud Detection Using Bayesian and Neural Networks. In: Proceedings of the 1st International NAISO Congress on Neuro Fuzzy Technologies, Havana, Cuba (2002)
22. Masters, T.: Practical neural network recipes in C++. Academic Press, London (1993)
23. Mena, J.: Investigative data mining for security and criminal detection. Butterworth-Heinemann (2003)
24. Mitchell, T.M.: Machine learning. MIT Press and The McGraw-Hill Companies, Inc. (1997)
25. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical recipes in C. Cambridge University Press, Cambridge (1992)
26. Robinson, A.J., Fallside, F.: Static and dynamic error propagation networks with application to speech coding. In: Anderson, D.Z. (ed.) Neural Information Processing System, American Institute of Physics (1998)
27. Williams, R.J., Zipser, D.: Gradient-based learning algorithms for recurrent networks and their computational complexity. Back-propagation: Theory, Architectures and Applications (1995)