Manuscript is intended for the *Special Issue: Neural Network Learning in Big Data*

HADOOP NEURAL NETWORK FOR PARALLEL AND DISTRIBUTED FEATURE SELECTION

Victoria J. Hodge, Simon O'Keefe & Jim Austin

Advanced Computer Architecture Group,
Department of Computer Science,
University of York, York, YO10 5GH, UK.

{victoria.hodge, simon.okeefe, jim.austin}@york.ac.uk

Corresponding Author:

Dr Victoria J. Hodge.
Dept of Computer Science,
University of York,
Deramore Lane,
York, UK
YO10 5GH
Email: Victoria.hodge@york.ac.uk
Phone: +44 (0)1904 325637
Fax: +44 (0)1904 325599

Abstract

In this paper, we introduce a theoretical basis for a Hadoop-based neural network for parallel and distributed feature selection in Big Data sets. It is underpinned by an associative memory (binary) neural network which is highly amenable to parallel and distributed processing and fits with the Hadoop paradigm. There are many feature selectors described in the literature which all have various strengths and weaknesses. We present the implementation details of five feature selection algorithms constructed using our artificial neural network framework embedded in Hadoop YARN. Hadoop allows parallel and distributed processing. Each feature selector can be divided into subtasks and the subtasks can then be processed in parallel. Multiple feature selectors can also be processed simultaneously (in parallel) allowing multiple feature selectors to be compared. We identify commonalities among the five features selectors. All can be processed in the framework using a single representation and the overall processing can also be greatly reduced by only processing the common aspects of the feature selectors once and propagating these aspects across all five feature selectors as necessary. This allows the best feature selector and the actual features to select to be identified for large and high dimensional data sets through exploiting the efficiency and flexibility of embedding the binary associative-memory neural network in Hadoop.

Keywords

Hadoop; MapReduce; Distributed; Parallel; Feature Selection; Binary Neural Network

1 Introduction

The meaning of "big" with respect to data is specific to each application domain and dependent on the computational resources available. Here we define "Big Data" as large, dynamic collections of data that cannot be processed using traditional techniques, a definition adapted from (Zikopoulos & Eaton, 2011; Franks, 2012). Today, data is generated continually by an increasing range of processes and in ever increasing quantities driven by Big Data mechanisms such as cloud computing and on-line services. Business and scientific data from many fields, such as finance, astronomy, bioinformatics and physics,

are often measured in terabytes (10^{12} bytes). Big Data is characterised by its complexity, variety, speed of processing and volume (Laney, 2001). It is increasingly clear that exploiting the power of these data is essential for information mining. These data often contain too much noise (Liu, Motada, Setiono & Zhao, 2010) for accurate classification (Dash & Liu, 1997; Han & Kamber, 2006), prediction (Dash & Liu, 1997; Guyon & Elisseeff, 2003) or outlier detection (Hodge, 2011). Thus, only some of the features (dimensions) are related to the target concept (classification label or predicted value). Also, if there are too many data features then the data points become sparse. If data is too sparse then distance measures such as the popular Euclidean distance and the concept of nearest neighbours become less applicable (Ertöz, Steinbach & Kumar, 2003). Many machine learning algorithms are adversely affected by this noise and these superfluous features in terms of both their accuracy and their ability to generalize. Consequently, the data must be pre-processed by the classification or prediction algorithm itself or by a separate feature selection algorithm to prune these superfluous features (Kohavi & John, 1997; Witten & Frank, 2000).

The benefits of feature selection include: reducing the data size when superfluous features are discarded, improving the classification/prediction accuracy of the underlying algorithm where the algorithm is adversely affected by noise, producing a more compact and easily understood data representation and reducing the execution time of the underlying algorithm due to the smaller data size. Reducing the execution time is extremely important for Big Data, which has a high computational resource demand on memory and CPU time.

In this paper, we focus on feature selection in vast data sets for parallel and distributed classification systems. We aim to remove noise and reduce redundancy to improve classification accuracy. There is a wide variety of techniques proposed in the machine learning literature for feature selection including Correlation-based Feature Selection (Hall, 1998), Principal Component Analysis (PCA) (Jolliffe, 2002), Information Gain (Quinlan, 1986), Gain Ratio (Quinlan, 1992), Mutual Information Selection (Wettscherek, 1994), Chi-square Selection (Liu & Setiono, 1995), Probabilistic Las Vegas Selection (Liu & Setiono, 1996) and Support Vector Machine Feature Elimination (Guyon, Weston, Barnhill & Vapnik, 2002). Feature selectors produce feature scores. Some feature selectors also select the best set of features to use while others just rank the features with the scores. For these feature rankers, the best set of features must then be chosen by the user, for example, using greedy search (Witten & Frank, 2000).

It is often not clear to the user which feature selector to use for their data and application. In their analysis of feature selection, Guyon and Elisseeff (2003) recommend evaluating a variety of feature selectors before deciding the best for their problem. Therefore, we propose that users exploit our framework to run a variety of feature selectors in parallel and then evaluate the feature sets chosen by each selector using their own specific criteria. Having multiple feature selectors available also provides the opportunity for ensemble feature selection where the results from a range of feature selectors are merged to generate the best set of features to use. Feature selection is a combinatorial problem so needs to be implemented as efficiently as possible particularly on big data sets. We have previously developed a k-NN classification (Weeks et al., 2003; Hodge & Austin, 2005) and prediction algorithm (Hodge, Krishnan, Austin & Polak, 2011) using an associative memory (binary) neural network called the Advanced Uncertain Reasoning Architecture (AURA) (Austin, 1995). This multi-faceted k-NN motivated a unified feature selection framework exploiting the speed and storage efficiency of the associative memory neural network. The framework lends itself to parallel and distributed processing across multiple nodes allowing vast data sets to be processed. This could be done by processing the data at the same geographical location using a single machine with multiple processing cores (Weeks, Hodge & Austin, 2002) or at the same geographical location using multiple compute nodes (Weeks, Hodge & Austin, 2002) or even distributed processing of the data at multiple geographical locations.

Data mining tools such as Weka (Witten and Frank, 2000), Matlab, R and SPSS provide feature selection algorithms for data mining and analytics. However, these products are designed for small scale data analysis. Researchers have parallelised individual feature selection algorithms using MapReduce/Hadoop (Chu et al., 2006; Reggiani, 2013; Singh et al., 2009; Sun, 2014). Data mining libraries such as Mahout (https://mahout.apache.org) and MLib (https://spark.apache.org/mllib/) and

data mining frameworks such as Radoop (https://rapidminer.com/products/radoop/) include a large number of data mining algorithms including feature selectors. However, they do not explicitly tackle processing reuse with a view to multi-user and multi-task resource allocation. Zhang. Kumar and Re (2014) developed a database systems framework for optimised feature selection providing a range of algorithms. They observed that there are reuse opportunities that could yield orders of magnitude performance improvements on feature selection workloads as we will also demonstrate here using AURA in an Apache Hadoop (https://hadoop.apache.org/) framework.

The main contributions of this paper are:

- To extend the AURA framework to parallel and distributed processing of vast data sets in Apache Hadoop,
- To describe five feature selectors in terms of the AURA framework. Two of the feature selectors have been implemented in AURA but not using Hadoop (Hodge, O'Keefe & Austin, 2006; Hodge, Jackson & Austin, 2012) and the other three have not been implemented in AURA before,
- To theoretically analyse the resulting framework to show how the five feature selectors have common requirements to enable reuse.
- To theoretically analyse the resulting framework to show how we reduce the number of computations. The larger the data set then the more important this reduction becomes.
- To demonstrate parallel and distributed processing in the framework allowing Big Data to be analysed.

In our AURA framework, the feature selectors all use one common data representation. We only need to process any common elements once and can propagate the common elements to all feature selectors that require them. Thus, we can rapidly and efficiently determine the best feature selector and the best set of features to use for each data set under investigation. In section 2, we discuss AURA and related neural networks and how to store and retrieve data from AURA, section 3 demonstrates how to implement five feature selection algorithms in the AURA unified framework and section 4 describes parallel and distributed feature selection using AURA. We than analyse the unified framework in section 5 to identify common aspects of the five feature selectors and how they can be implemented in the unified framework in the most efficient way. Section 6 details the overall conclusions from our implementations and analyses.

2 Binary Neural Networks

AURA (Austin, 1995) is a hetero-associative memory neural network (Palm, 2013). An associative memory is addressable through its contents and a hetero-associative memory stores associations between input and output vectors where the vectors are different (Palm, 2013). AURA uses binary Correlation Matrix Memories (CMMs): binary hetero-associative matrices that store and retrieve patterns using matrix calculus. They are non-recursive and fully connected. Input vectors (stimuli) address the CMM rows and output vectors address the CMM columns. Binary neural networks have a number of advantages compared to standard neural networks including rapid one-pass training, high levels of data compression, computational simplicity, network transparency, a partial match capability and a scalable architecture that can be easily mapped onto high performance computing platforms including parallel and distributed platforms (Weeks, Hodge & Austin, 2002). AURA is implemented as a C++ software library.

Previous parallel and distributed applications of AURA have included distributed text retrieval (Weeks, Hodge & Austin, 2002), distributed time-series signal searching (Fletcher, Jackson, Jessop, Liang, & Austin, 2006) and condition monitoring (Austin, Brewer, Jackson & Hodge, 2010). This new development will augment these existing techniques and is aimed at these same domains. It will couple feature selection, classification and prediction with the speed and storage efficiency of a binary neural network allowing parallel and distributed data mining. This makes AURA ideal to use as the basis of an

efficient distributed machine learning framework. A more formal definition of AURA, its components and methods now follows.

2.1 AURA

The AURA methods use binary input $\boldsymbol{I}$ and output $\boldsymbol{O}$ vectors to efficiently store records in a CMM $\boldsymbol{M}$ as in equation 1 using the binary rule (Palm, 2013).

$$\boldsymbol{M} = \bigvee \boldsymbol{I_j O_j^T} \ \ where \ \vee \ is \ logical \ OR \tag{1}$$

Training (construction of a CMM) is a single epoch process with one training step for each input-output association (each $\boldsymbol{I_j\, O_j^T}$ in equation 1) which equates to one step for each record $\boldsymbol{j}$ in the data set. Thus, the trained CMM $\boldsymbol{M}$ represents $\{(\boldsymbol{I_1}\times\boldsymbol{O_1^T}), (\boldsymbol{I_2}\times\boldsymbol{O_2^T}), \ldots (\boldsymbol{I_n}\times\boldsymbol{O_n^T})\}$ superimposed using bitwise ***or***. $\boldsymbol{I_j\, O_j^T}$ is an estimate of the weight matrix $\boldsymbol{W(j)}$ of the synaptic connections of the neural network as a linear associator with binary weights. $\boldsymbol{W(j)}$ forms a mapping representing the association described by the *j*th input/output pair of vectors. As a consequence of using unipolar elements {0, 1} throughout, the value at each matrix component $\boldsymbol{w_{ij}}$ means the existence of an association between elements $\boldsymbol{i}$ and $\boldsymbol{j}$. The trained CMM $\boldsymbol{M}$ is then effectively an encoding (correlation) of the $\boldsymbol{N}$ weight matrices $\boldsymbol{W}$ for all $\boldsymbol{N}$ records in the data set. Individual weights within the weight matrix update using a generalisation of Hebbian learning (Hebb, 1949) where the state for each synapse (matrix element) is binary valued. Every synapse can update its weight independently using a *local learning rule* (Palm, 2013). Local learning is biologically plausible and computationally simple allowing parallel and rapid execution. The learning process is illustrated in Figure 1.

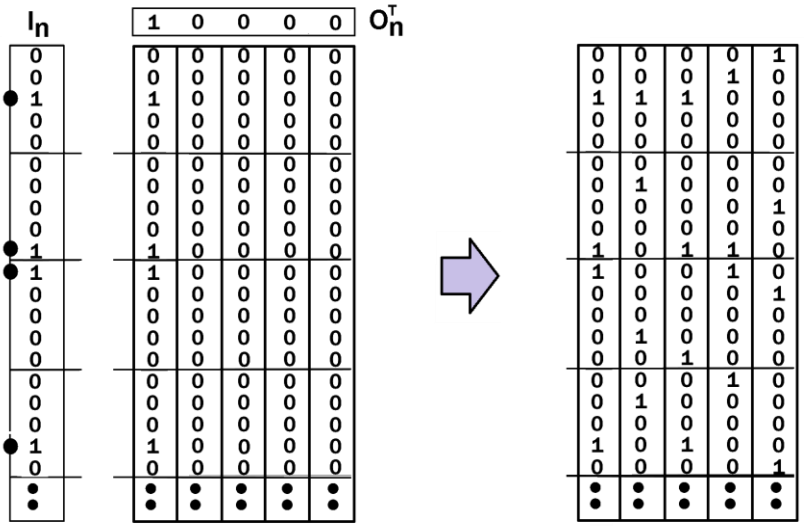

Figure 1 Showing a CMM learning input vector I_n associated with output vector O_n on the left. The CMM on the right shows the CMM after five associations $I_j\, O_j^T$. Each column of the CMM represents a record. Each row represents a feature value for qualitative features or a quantisation of feature values for quantitative features and each set of rows (shown by the horizontal lines) represents the set of values or set of quantisations for a particular feature.

For feature selection, the data are stored in the CMM which forms an index of all features in all records. During training, the input vectors $\boldsymbol{I_j}$ represent the feature and class values and are associated with a unique output vector $\boldsymbol{O_j}$ representing a record. Figure 1 shows a trained CMM. In this paper, we set only one bit in the vector $\boldsymbol{O_j}$ indicating the location of the record in the data set, the first record has the first bit set, the second record has the second bit set etc. Using a single set bit makes the length of $\boldsymbol{O_j}$ potentially large. However, exploiting a compact list representation (Hodge & Austin, 2001) (more detail is provided in section 4.3.1) means we can compress the storage representation.

2.2 Data

The AURA feature selector, classifier and predictor framework can handle qualitative features (symbolic and discrete numeric) and quantitative features (continuous numeric).

The raw data sets need pre-processing to allow them to be used in the binary AURA framework. Qualitative features are enumerated and each separate token maps onto an integer (***Token*** ↦***Integer***) which identifies the bit to set within the vector. For example, a SEX_TYPE feature would map as (***F*** ↦ ***0***) and (***M*** ↦ ***1***). Any quantitative features are quantised (mapped to discrete bins) (Hodge & Austin, 2012). Each individual bin maps onto an integer which identifies the bit to set in the input vector. Next, we describe the simple equi-width quantisation. We note that the Correlation-Based Feature Selector described in section 3.2 uses a different quantisation technique to determine the bin boundaries. However, once the boundaries are determined, the mapping to CMM rows is the same as described here.

To quantise quantitative features, a range of input values for feature $\boldsymbol{F_f}$ map onto each bin. Each bin maps to a unique integer as in equation 2 to index the correct location for the feature in $\boldsymbol{I_j}$. In this paper, the range of feature values mapping to each bin is equal to subdivide the feature range into ***b*** equi-width bins across each feature.

$$\Re_{f_i} \rightarrow \boldsymbol{bins}_{f_k} \mapsto \boldsymbol{Integer}_{f_k} + \boldsymbol{offset}(\boldsymbol{F_f}) \quad (2)$$

where $\boldsymbol{F_f} \in \boldsymbol{F}, \boldsymbol{f_i} \textit{is a value of } \boldsymbol{F_f}$ and $\boldsymbol{cardinality}(\boldsymbol{Integer}_{f_k}) \equiv \boldsymbol{cardinality}(\boldsymbol{bins}_{f_k})$

In equation 2, ***offset***($\boldsymbol{F_f}$) is a cumulative integer offset within the binary vector for each feature $\boldsymbol{F_f}$, → is a many-to-one mapping and ↦ is a one-to-one mapping. The offset for the next feature $\boldsymbol{F_{f+1}}$ is given by ***offset***($\boldsymbol{F_{f+1}}$) = ***offset***($\boldsymbol{F_f}$) + ***nBins***($\boldsymbol{F_f}$) where ***nBins***($\boldsymbol{F_f}$) is the number of bins for feature $\boldsymbol{F_f}$.

For each record in the data set
 For each feature
  Calculate bin for feature value;
  Set bit in vector as in equation 2;

2.3 AURA Recall

To recall the matches for a query (input) record, we firstly produce a recall input vector $\boldsymbol{R_k}$ by quantising the target values for each feature to identify the bins (CMM rows) to activate as in equation 3. During recall, the presentation of recall input vector $\boldsymbol{R_k}$ elicits the recall of output vector $\boldsymbol{O_k}$ as vector $\boldsymbol{R_k}$ contains all of the addressing information necessary to access and retrieve vector $\boldsymbol{O_k}$. Recall is effectively the dot product of the recall input vector $\boldsymbol{R_k}$ and CMM ***M***, as in equation 3 and Figure 2.

$$\boldsymbol{S^T} = \boldsymbol{R_k^T} \cdot \boldsymbol{M} \quad (3)$$

If $\boldsymbol{R_k}$ appeared in the training set, we get an integer-valued vector ***S*** (the summed output vector), composed of the required output vector multiplied by a weight based on the dot product of the input vector with itself. If the recall input $\boldsymbol{R_k}$ is not from the original training set, then the system will recall the output $\boldsymbol{O_k}$ associated with the closest stored input to $\boldsymbol{R_k}$, based on the dot product between the test and training inputs.

Matching is a combinatorial problem but can be achieved in a single pass in AURA. AURA can also exploit the advantages of sparse vectors (Palm, 2013) during recall by only activating regions of interest. If the input vector $\boldsymbol{R_k}$ has 1,000 bits indexing 1,000 CMM rows then only the rows addressed by a set bit in the input vector need be examined (as shown in figures 2 and 3). For a 10 bit set vector then only 10 of the 1,000 rows are activated. The input pattern $\boldsymbol{R_k}$ would be said to have a saturation of (10/1000 = 0.01). The total amount of data that needs to be examined is reduced by a factor that is dependent on this saturation providing that the data is spread reasonably evenly between the rows and the CMM is implemented effectively. Using smart encoding schemes can bring the performance

improvement resulting from very low saturation input patterns to over 100-fold (Weeks, Hodge & Austin, 2002).

The AURA technique thresholds the summed output $\boldsymbol{S}$ to produce a binary output vector $\boldsymbol{T}$ as given in equation 4.

$$\boldsymbol{T_j} = \begin{cases} 1 \text{ if } \boldsymbol{S_j} \geq \theta \\ 0 \text{ otherwise} \end{cases} \quad (4)$$

For exact match, we use the Willshaw threshold (Willshaw, Buneman & Longuet-Higgins, 1969) to set θ. This sets a bit in the thresholded output vector for every location in the summed output vector that has a value higher than or equal to θ. The value of θ varies according to the task. If there are ten features in the data and we want to find all stored records that match the ten feature values of the input vector then we set θ to 10. Thus, for full match $\theta = b^1$, where b^1 is set to the number of set bits in the input vector. For partial matching, we use the L-Max threshold (Casasent & Telfer, 1992). L-Max thresholding essentially retrieves *at least L* top matches. Our AURA software library automatically sets θ to the highest integer value that will retrieve at least $\boldsymbol{L}$ matches.

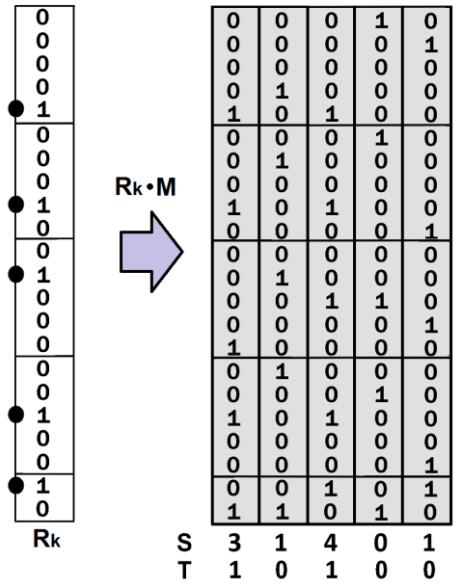

Figure 2 Showing a CMM recall. Applying the recall input vector R_k to the CMM M retrieves a summed integer vector S with the match score for each CMM column. S is then thresholded to retrieve the matches. The threshold here is either Willshaw with value 3 retrieving all columns that sum to 3 or more or L-Max with value 2 to retrieve the 2 highest scoring columns.

Feature selection described in section 3 requires both exact matching using Willshaw thresholding and partial matching using L-Max thresholding.

3 Feature Selection

There are two fundamental approaches to feature selection (Kohavi & John, 1997; Witten & Frank, 2000): (1) filters select the optimal set of features independently of the classifier/predictor algorithm while (2) wrappers select features which optimise classification/prediction using the algorithm. We examine the mapping of five filter approaches to the binary AURA architecture. Filter approaches are more flexible than wrapper approaches as they are not directly coupled to the algorithm and are thus applicable to a wide variety of classification and prediction algorithms. Our method exploits the high speed and efficiency of the AURA techniques as feature selection is a combinatorial problem.

We examine a mutual information approach (**Mutual Information Feature Selection (MI)** detailed in section 3.1 that analyses features on an individual basis, a correlation-based multivariate filter approach (**Correlation-based Feature Subset Selection (CFS)** detailed in section 3.2 that examines greedily selected subsets of features, a revised Information Gain approach **Gain Ratio (GR)** detailed in section 3.3, a feature dependence approach **Chi-Square Feature selection(CS)** detailed in section 3.4 which is univariate, and a univariate feature relevance approach **Odds Ratio (OR)** detailed in section 3.5.

Univariate filter approaches such as MI, CS or OR are quicker than multivariate filters as they do not need to evaluate all combinations of subsets of features. The advantage of a multivariate filter compared to a univariate filter lies in the fact that a univariate approach does not account for interactions between features. Multivariate techniques evaluate the worth of feature subsets by considering both the individual predictive ability of each feature and the degree of redundancy between the features in the set.

All five feature selection algorithms have their relative strengths. We refer the reader to Forman (2003) and Varela et al. (2013) for accuracy evaluations of these feature selectors. These papers show that the best feature selector varies with data and application. Using the CFS attribute selector, Hall and Smith (1998) found significant improvement in classification accuracy of k-NN on five of the 12 data sets they evaluated but a significant degradation in accuracy on two data sets. Hence, different feature selectors are required for different data sets and applications.

We note that the CFS as implemented by Hall (1998) uses an entropy-based quantisation whereas we have used equi-width quantisation for the other feature selectors (MI, GR, CS and OR). We plan to investigate unifying the quantisation as a next step. For the purpose of our analysis in section 5, we assume that all feature selectors are using identical quantisation. We assume that all records are to be used during feature selection**.**

3.1 Mutual Information Feature Selection

Wettscherek (1994) described a mutual information feature selection algorithm. The mutual information between two features is ``*the reduction in uncertainty concerning the possible values of one feature that is obtained when the value of the other feature is determined*'' (Wettscherek, 1994). MI is defined by equation 5:

$$\boldsymbol{MI(F_j, C)} = \sum_{i=1}^{b(F_j)} \sum_{c=1}^{nClass} \boldsymbol{p(C = c \wedge F_j = f_i) \cdot log_2 \left(\frac{p(C = c \wedge F_j = f_i)}{p(C = c) \cdot p(F_j = f_i)} \right)} \quad (5)$$

To calculate p($\boldsymbol{C = c \wedge F_j = f_i}$), we use AURA to calculate $\frac{n(BVf_i \wedge BVc)}{N}$.

AURA excites the row in the CMM corresponding to feature value $\boldsymbol{f_i}$ of feature $\boldsymbol{F_j}$ and the row in the CMM corresponding to class value $\boldsymbol{c}$ as shown in Figure 3. By thresholding the output vector $\boldsymbol{S}$ at Willshaw threshold value = 2, we obtain a thresholded output vector with a bit set for every co-occurrence. We can count these set bits to determine the co-occurrence count. Furthermore, $p(C = c)$is the count of the number of set bits ***n(BVc)*** in the binary vector (CMM row) for $\boldsymbol{c}$ and $p(F_j = f_i)$ is the count of the number of set bits ***n(BVf$_i$)*** in the binary vector (CMM row) for $\boldsymbol{f_i}$ as used by GR.
The MI calculated using AURA for qualitative features is given by equation 6 where $\boldsymbol{N}$ is the number of records in the data set, ***rows(F$_j$)*** is the number of CMM rows for feature $\boldsymbol{F_j}$ and ***nClass*** is the number of classes:

$$\boldsymbol{MI(F_j, C)} = \sum_{i=1}^{rows(F_j)} \sum_{c=1}^{nClass} \boldsymbol{\frac{n(BVf_i \wedge BVc)}{N} \cdot log_2 \left(\frac{\frac{n(BVf_i \wedge BVc)}{N}}{\frac{n(BVf_i)}{N} \cdot \frac{n(BVc)}{N}} \right)} \quad (6)$$

We can follow the same process for real/discrete ordered numeric features in AURA. In this case, the mutual information is given by equation 7:

$$MI(F_j, C) = \sum_{i=1}^{bins(F_j)} \sum_{c=1}^{nClass} \frac{n(BVb_i \wedge BVc)}{N} \cdot log_2 \left(\frac{\frac{n(BVb_i \wedge BVc)}{N}}{\frac{n(BVb_i)}{N} \cdot \frac{n(BVc)}{N}} \right) \quad (7)$$

where ***bins***(F_j) is the number of bins (effectively the number of rows) in the CMM for feature F_j and BVb_i is the CMM row for the bin mapped to by feature value f_i,

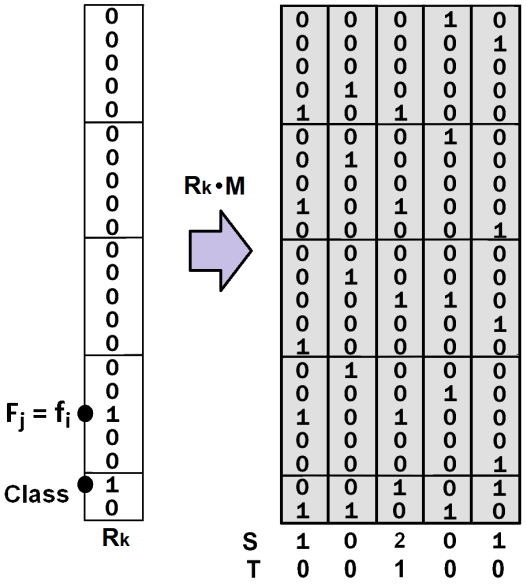

Figure 3 Diagram showing the feature value row and the class values row excited to determine co-occurrences ($C = c \wedge F_j = f_i$).

The MI feature selector assumes independence of features and scores each feature separately so it is the user's prerogative to determine the number of features to select. The major drawback of the MI feature selector along with similar information theoretic approaches, for example Information Gain, is that they are biased toward features with the largest number of distinct values as this splits the training records into nearly pure classes. Thus, a feature with a distinct value for each record has a maximal information score. The CFS and GR feature selectors make adaptations of information theoretic approaches to prevent this biasing.

3.2 Correlation-based Feature Subset Selection

Hall (1998) proposed the Correlation-based Feature Subset Selection (CFS). It measures the strength of the correlation between pairs of features. CFS favours feature subsets that contain features that are highly correlated to the class but uncorrelated to each other to minimise feature redundancy. CFS is thus based on information theory measured using Information Gain. Hall and Smith (1997) used a modified Information Gain measure, Symmetrical Uncertainty, (***SU***) given in equation 8 to prevent bias towards features with many distinct values (section 3.1). ***SU*** estimates the correlation between features by normalising the value in the range [0, 1]. Two features are completely independent if ***SU=0*** and completely dependent if ***SU=1***.

$$SU(F_j, G_l) = 2.0 \cdot \left[\frac{Ent(F_j) - Ent(F_j \mid G_l)}{Ent(F_j) + Ent(G_l)} \right] \quad (8)$$

where the entropy of a feature F_j for all feature values f_i is given as equation 9:

$$Ent(F_j) = -\sum_{i=1}^{n(F_j)} p(f_i) log_2(p(f_i)) \quad (9)$$

and the entropy of feature $\boldsymbol{F_j}$ after observing values of feature $\boldsymbol{G_l}$ is given as equation 10:

$$\boldsymbol{Ent(F_j \mid G_l) = -\sum_{k=1}^{n(G_l)} p(g_k) \sum_{i=1}^{n(F_j)} p(f_i \mid g_k) log_2(p(f_i \mid g_k))} \quad (10)$$

Any quantitative features are discretised using Fayyad and Irani's entropy quantisation (Fayyad & Irani, 1993). The bin boundaries are determined using Information Gain and these quantisation bins map the data into the AURA CMM as previously.

CFS has many similarities to MI when calculating the values in equations 8, 9 and 10 and through using the same CMM (Figure 3) as noted below.

In the AURA CFS, for each pair of features ($\boldsymbol{F_j}$,$\boldsymbol{G_l}$) to be examined, the CMM is used to calculate ***Ent***($\boldsymbol{F_j}$), ***Ent***($\boldsymbol{G_l}$) and ***Ent***($\boldsymbol{F_j \mid G_l}$) from equations 8, 9 and 10. There are three parts to the calculation.

1. ***Ent***($\boldsymbol{F_j}$) requires the count of data records for the particular value $\boldsymbol{f_i}$ of feature $\boldsymbol{F_j}$ which is $\boldsymbol{n(BVf_i)}$ in equation 6 for qualitative and class features and $\boldsymbol{n(BVb_i)}$ in equation 7 for quantitative features. AURA excites the row in the CMM corresponding to feature value $\boldsymbol{f_i}$ of feature $\boldsymbol{F_j}$. This row is a binary vector (BV) and is represented by $\boldsymbol{BVf_i}$. A count of bits set on the row gives $\boldsymbol{n(BVf_i)}$ from equation 6 and is achieved by thresholding the output vector $\boldsymbol{S_k}$ from equation 4 at Willshaw value 1.
2. Similarly, ***Ent***($\boldsymbol{G_l}$) counts the number of records where feature $\boldsymbol{G_l}$ has value $\boldsymbol{g_k}$.
3. ***Ent***($\boldsymbol{F_j \mid G_l}$) requires the number of co-occurrences of a particular value $\boldsymbol{f_i}$ of feature $\boldsymbol{F_j}$ with a particular value $\boldsymbol{g_k}$ of feature $\boldsymbol{G_l}$ $\boldsymbol{n(BVf_i \wedge BVg_k)}$ for qualitative features and $\boldsymbol{n(BVb_i \wedge BVb_k)}$ for quantitative features and between a feature and the class $\boldsymbol{n(BVf_i \wedge BVc)}$ and $\boldsymbol{n(BVb_i \wedge BVc)}$ for qualitative and quantitative features respectively. If both the feature value row and the class values row are excited then the summed output vector will have a two in the column of every record with a co-occurrence of $\boldsymbol{f_i}$ with $\boldsymbol{c_j}$ as shown in Figure 3. By thresholding the summed output vector at a threshold of two, we can find all co-occurrences. We represent this number of bits set in the vector by $\boldsymbol{n(BVf_i \wedge BVc)}$ which is a count of the set bits when $\boldsymbol{BVc}$ is logically *and*ed with $\boldsymbol{BVf_i}$.

CFS determines the feature subsets to evaluate using forward search. Forward search works by greedily adding features to a subset of selected features until some termination condition is met whereby adding new features to the subset does not increase the discriminatory power of the subset above a pre-specified threshold value. The major drawback of CFS is that it cannot handle strongly interacting features (Hall & Holmes, 2003).

3.3 Gain Ratio Feature Selection

Gain Ratio (GR) (Quinlan, 1992) is a new feature selector for the AURA framework. GR is a modified Information Gain technique and is used in the popular machine learning decision tree classifier C4.5 (Quinlan, 1992). Information Gain is given in equation 11 for feature $\boldsymbol{F_j}$ and the class $\boldsymbol{C}$. CFS (section 3.2) modifies Information Gain to prevent biasing toward features with the most values. GR is an alternative adaptation which considers the number of splits (number of values) of each feature when calculating the score for each feature using normalisation.

$$\boldsymbol{Gain(F_j, C) = Ent(F_j) - Ent(F_j \mid C)} \quad (11)$$

where ***Ent***($\boldsymbol{F_j}$) is defined in equation 9 and ***Ent***($\boldsymbol{F_j \mid C}$) is defined by equation 10. Then Gain Ratio is defined as equation 12:

$$\boldsymbol{GainRatio(F_j, C) = \frac{Gain(F_j, C)}{IntrinsicValue(F_j)}} \quad (12)$$

where ***IntrinsicValue*** is given by equation 13:

$$\boldsymbol{IntrinsicValue}(\boldsymbol{F_j}) = \sum_{\boldsymbol{p=1}}^{\boldsymbol{V}} \frac{\boldsymbol{S_p}}{\boldsymbol{N}} \boldsymbol{log_2}\left(\frac{\boldsymbol{S_p}}{\boldsymbol{N}}\right) \tag{13}$$

and $\boldsymbol{V}$ is the number of feature values $\boldsymbol{(n(F_j))}$ for qualitative features and number of quantisation bins $\boldsymbol{n(b_i)}$ for quantitative features and $\boldsymbol{S_p}$ is a subset of the records that have $\boldsymbol{F_j{=}f_i}$ for qualitative features or map to the quantisation bin $\boldsymbol{bin(f_i)}$ for quantitative features.

To implement GR using AURA, we train the CMM as described in section 2.1 We can then calculate $\boldsymbol{Ent(F_j)}$ and $\boldsymbol{Ent(F_j \mid C)}$ as per the CFS feature selector described in section 3.2 to allow us to calculate $\boldsymbol{Gain(F_j, C)}$. To calculate $\boldsymbol{IntrinsicValue(F_j)}$ we need to calculate the number of records that have particular feature values. This is achieved by counting the number of set bits $\boldsymbol{n(BVf_i)}$ in the binary vector (CMM row) for $\boldsymbol{f_i}$ for qualitative features or $\boldsymbol{n(BVb_i)}$ in the binary vector for the quantisation bin $\boldsymbol{b_i}$ for quantitative features. We can store counts for the various feature values and classes as we proceed so there is no need to calculate any count more than once.

The main disadvantage of GR is that it tends to favour features with low Intrinsic Value rather than high gain by overcompensating toward a feature just because its intrinsic information is very low.

3.4 Chi-Square Algorithm

We now demonstrate how to implement a second new feature selector in the AURA framework. The Chi-Square (CS) (Liu & Setiono, 1995) algorithm is a feature ranker like MI, OR and GR rather than a feature selector; it scores the features but it is the user's prerogative to select which features to use. CS assesses the independence between a feature ($\boldsymbol{F_j}$) and a class ($\boldsymbol{C}$) and is sensitive to feature interactions with the class. Features are independent if CS is close to zero. Yang and Pedersen (1997) and Forman (2003) conducted evaluations of filter feature selectors and found that CS is among the most effective methods of feature selection for classification.

Chi-Square is defined as equation 14:

$$\boldsymbol{\chi^2(F_j, C)} = \sum_{\boldsymbol{i=1}}^{\boldsymbol{b(F_j)}} \sum_{\boldsymbol{c=1}}^{\boldsymbol{nClass}} \frac{\boldsymbol{N * (wz - yx)^2}}{\boldsymbol{(w+y) * (x+z) * (w+x) * (y+z)}} \tag{14}$$

where $\boldsymbol{b(F_j)}$ is the number of bins (CMM rows) representing feature $\boldsymbol{F_j}$, $\boldsymbol{nClass}$ is the number of classes, $\boldsymbol{w}$ is the number of times $\boldsymbol{f_i}$ and $\boldsymbol{c}$ co-occur, $\boldsymbol{x}$ is the number of times $\boldsymbol{f_i}$ occurs without $\boldsymbol{c}$, $\boldsymbol{y}$ is the number of times $\boldsymbol{c}$ occurs without $\boldsymbol{f_i}$, $\boldsymbol{z}$ is the number of times neither $\boldsymbol{c}$ nor $\boldsymbol{f_i}$ occur. Thus, CS is predicated on counting occurrences and co-occurrences and, hence, has many commonalities with MI, CFS and GR.

- Figure 3 shows how to produce a binary output vector ($\boldsymbol{BVf_i \wedge BVc}$) for qualitative features or ($\boldsymbol{BVb_i \wedge BVc}$) for quantitative features listing the co-occurrences of a feature value and a class value. It is then simply a case of counting the number of set bits (1s) in the thresholded binary vector $\boldsymbol{T}$ in Figure 3 to count $\boldsymbol{w}$.
- To count $\boldsymbol{x}$ for qualitative features, we logically subtract ($\boldsymbol{BVf_i \wedge BVc}$) from the binary vector ($\boldsymbol{BVf_i}$) to produce a binary vector and count the set bits in the resulting vector. For quantitative features, we subtract ($\boldsymbol{BVb_i \wedge BVc}$) from ($\boldsymbol{BVb_i}$) and count the set bits in the resulting binary vector.
- To count $\boldsymbol{y}$ for qualitative features, we can logically subtract ($\boldsymbol{BVf_i \wedge BVc}$) from ($\boldsymbol{BVc}$) and count the set bits and likewise for quantitative features we can subtract ($\boldsymbol{BVb_i \wedge BVc}$) from $\boldsymbol{BVc}$ and count the set bits.
- If we logically *or* ($\boldsymbol{BVf_i}$) with ($\boldsymbol{BVc}$), we get a binary vector representing $\boldsymbol{(F_j{=}f_i) \vee (C{=}c)}$ for qualitative features. For quantitative features, we can logically *or* ($\boldsymbol{BVb_i}$) with ($\boldsymbol{BVc}$) to produce

$(F_j=bin(f_i)) \vee (C=c)$. If we then logically invert this new binary vector, we retrieve a binary vector representing z and it is simply a case of counting the set bits to get the count for z.

As with MI and OR, CS is univariate and assesses features on an individual basis selecting the features with the highest scores, namely the features that interact most with the class.

3.5 Odds Ratio

The third new feature selector is Odds Ratio (OR) (see Forman, 2003). OR is another feature ranker. Standard OR is a two-class feature ranker although it can be extended to multiple classes. It is often used in text classification tasks as these are often two-class problems. It performs well particularly when used with Naïve Bayes Classifiers. OR reflects relevance as the likelihood (odds) of a feature occurring in the positive class normalized by that of the negative class. OR has many commonalities with MI, CFS and GR but particularly with CS where it requires the same four calculations *w*, *x*, *y* and *z* (defined above in section 3.4). Odds Ratio is defined by equation 15:

$$OR(F_j, C) = \sum_{i=1}^{b(F_j)} \frac{wz}{yx} \tag{15}$$

where $b(F_j)$ is the number of bins (CMM rows) representing feature F_j, w is the number of times f_i and c co-occur, x is the number of times f_i occurs without c, y is the number of times c occurs without f_i, z is the number of times neither c nor f_i occur. Thus, OR is predicated on counting occurrences and co-occurrences. To avoid division by zero the denominator is set to 1 if $\underline{yx}$ evaluates to 0.

4 Parallel and Distributed AURA

Feature selection is a combinatorial problem so a fast, efficient and scalable platform will allow rapid analysis of large and high dimensional data sets. AURA has demonstrated superior training and recall speed compared to conventional indexing approaches (Hodge & Austin, 2001) such as hashing or inverted file lists which may be used for data indexing. AURA trains 20 times faster than an inverted file list and 16 times faster than a hashing algorithm. It is up to 24 times faster than the inverted file list for recall and up to 14 times faster than the hashing algorithm. AURA k-NN has demonstrated superior speed compared to conventional k-NN (Hodge & Austin, 2005) and does not suffer the limitations of other k-NN optimisations such as the KD-tree which only scales to low dimensionality data sets (McCallum, Nigam & Ungar, 2000). We showed in (Hodge, O'Keefe & Austin, 2006) that using AURA speeds up the MI feature selector by over 100 times compared to a standard implementation of MI.

For very large data sets, the data may be processed in parallel on one compute node (such as a multi-core CPU) or across a number of distributed compute nodes. Each compute node in a distributed system can itself perform parallel processing.

4.1 Parallel AURA

In Weeks, Hodge & Austin (2002), we demonstrated a parallel search implementation of AURA. AURA can be subdivided across multiple processor cores within a single machine or spread across multiple connected compute nodes. This parallel processing entails "striping" the CMM across several parallel subsections. The CMM is effectively subdivided vertically across the output vector as shown in Figure 4. In the data, the number of features m is usually much less than the number of records N, $m << N$. Therefore, we subdivide the data along the number of records N (column stripes) as shown in the leftmost example in Figure 4.

Splitting the data across multiple CMM stripes using columns means that the CMM can store data as separate rows within a single stripe. Each record is contained within a single stripe. Each separate CMM stripe outputs a thresholded vector from that CMM stripe.

If the number of features is large then it is possible to subdivide the CMMs further. The CMM is divided vertically by the records (column stripes) as before and then the column stripes are subdivided by the

input features (row stripes). Subdivision by input features (row stripes) is shown in the rightmost diagram in figure 4. Dividing the CMM using the features (row stripes) makes assimilating the results more complex than assimilating the results for column stripes. Each row stripe produces a summed output vector containing column subtotals for those features within the stripe. The column subtotals need to be assimilated from all row stripes that hold data for that column. Thus, we sum these column subtotals to produce a column stripe vector $\boldsymbol{C}$ holding the overall sum for each column in that stripe. Row striping involves assimilating integer vectors of length $\boldsymbol{c}$ where $\boldsymbol{c}$ is the number of columns for the column subdivision (column stripe).

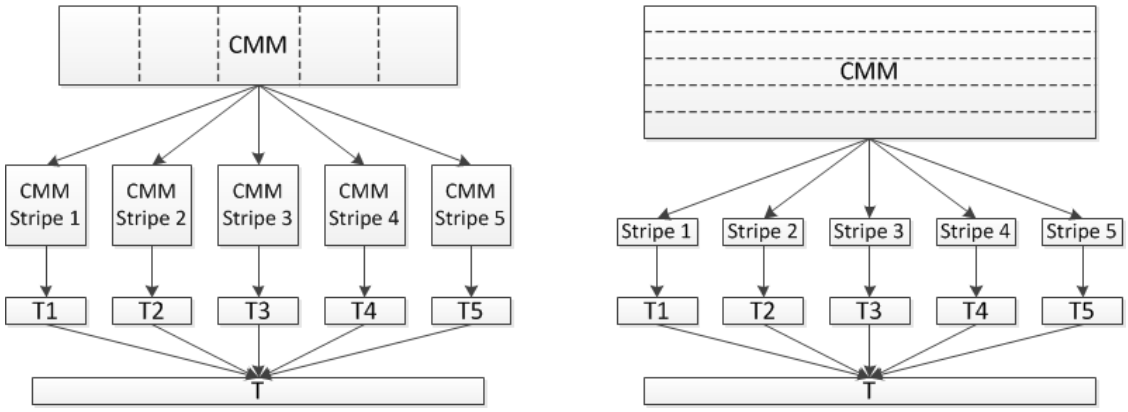

Figure 4 If a CMM contains large data it can be subdivided (striped) across a number of CMM stripes. In the left hand figure, the CMM is striped vertically (by time) and in the right hand figure the CMM is striped horizontally (be feature subsets). On the left, each CMM stripe produces a thresholded output vector T_n containing the top k matches (and their respective scores) for that stripe. All $\{T_n\}$ are aggregated to form a single output vector T which is thresholded to list the top matches overall. On the right, each stripe outputs a summed output vector S_n. All S_n are summed to produce an overall summed output vector which is thresholded to list the top matches overall.

4.2 Distributed AURA

There are two central challenges for distributed feature selection: firstly, maintaining a distributed data archive so that data does not have to be moved to a central repository and secondly, orchestrating the search process across the distributed data. Different data and applications will have different criteria that they wish to optimise. These could be optimising communication overhead, processing speed, memory usage or combinations of these criteria. Hence, there is unlikely to be a single best technique for distribution.

To distribute AURA, we use the striping mechanisms detailed in the previous section. However, rather than spreading the stripes within the cores of a multicore processor, we distribute the stripes across computers within a distributed network. The stripes need to be distributed for maximum efficiency. This can be to maximise processing speed, to minimise memory usage, to minimise communication overhead or a combination of criteria. Distributing the stripes requires an efficient distribution mechanism to underpin the procedure.

Orchestrated search with minimal data movement is provided by the open source software project: Apache Hadoop (Shvachko, Hairong, Radia & Chansler, 2010). Hadoop operates on the premise that "*moving computation is cheaper than moving data*" (Borthakur, 2008). Hadoop allows the distributed processing of large data sets across clusters of commodity servers. It provides load balancing, is highly scalable and has a very high degree of fault tolerance. It is able to run on commodity hardware due to its ability to detect and handle failures at the application layer. There are multiple copies of the stored data so, if one server or node is unavailable, its data can be automatically replicated from a known good copy. If a compute node fails then Hadoop automatically re-balances the work load on the remaining nodes. Hadoop has demonstrated high performance for a wide variety of tasks (Borthakur et al., 2011).

It was initially aimed at batch processing tasks so is ideally suited to the task of feature selection where the feature selector is trained with the training data and feature selection is run once on a large batch of test data. Hadoop is currently developing real-time processing capabilities. In this paper, we focus on batch processing and the implementation details of the five feature selectors using AURA with Hadoop.

Hadoop is highly configurable and can be optimised to the user's specific requirements, for example, optimising to minimise memory overhead, optimising for fastest processing or optimising to reduce communication overhead. Hence, we do not attempt to evaluate Hadoop here. Instead, we focus on describing how to map AURA CMMs to Hadoop to create a feature evaluation framework.

There are two parts of Hadoop that we consider here: YARN which assigns work to the nodes in a cluster and the Hadoop Distributed File System (HDFS) which is a distributed file system spanning all the nodes in the Hadoop cluster with a single namespace.

YARN (Kumar et al., 2013) supersedes MapReduce in Hadoop. YARN is able to run existing MapReduce applications. YARN decouples resource management and scheduling from the data processing. This means that data can continue to be streamed into the system simultaneously with MapReduce batch jobs. YARN has a central resource manager that reconciles Hadoop system resources according to constraints such as queue capacities or user-limits. Node manager agents monitor the processing operations of individual nodes in the cluster. The processing is controlled by an ApplicationMaster which negotiates resources from the central resource manager and works with the node manager agents to execute and monitor the tasks. The actual MapReduce procedure, divides (maps) the processing into separate chunks which are processed in parallel. The outputs of the processing tasks are combined (reduced) to generate a single result. The input and output data for MapReduce can be stored in HDFS on the same compute nodes used for processing the MapReduce jobs. This produces a very high aggregate bandwidth across the cluster. The user's applications specify the input/output locations and supply *map* and *reduce* functions via implementations of appropriate interfaces and/or abstract-classes. The framework takes care of distributing the software/configuration, scheduling tasks, monitoring the tasks and re-executing any failed tasks.

HDFS links together the file systems on many local nodes to make them into one big file system. HDFS assumes nodes will fail, so it achieves reliability by replicating data across multiple nodes. Processing data in situ on local nodes is efficient compared to moving the data over the network to a single processing node. This local processing architecture of Hadoop has resulted in very good performance (Rutman, 2011) on cheap computer clusters even with relatively slow network connections (such as 1 Gig Ethernet) (Rutman, 2011). Hence, Hadoop is ideal to underpin our distributed processing architecture.

4.3 Hadoop Feature Selection

Feature selection is a two part procedure. A training phase described in section 2.1 trains the data into the CMMs. A test phase then applies test data to the trained CMMs and correlates the results to produce feature selections. Each compute node holds a CMM, CMM stripe or set of CMM stripes that stores all local data. During training, CMMs are not immutable as each association in equation 1 changes the underlying CMM so Hadoop MapReduce is not a suitable paradigm for CMM training. Hence, the CMMs are trained in a conventional fashion and uploaded to HDFS once trained. If the data stored in a node's CMM exceed the memory capacity of that node then the CMM is subdivided into stripes as described in section 4.1 and shown in figures 4 and 5. The set of all CMM stripes at a node stores all data for that node. Every CMM stripe across the distributed system has to be coordinated so that record identifiers (such as timestamps) are matched to allow the CMM sum and threshold. Sum and threshold is column-based and relies on columns representing the same datum. When the results from different CMMs are unified then the columns from the various CMMs need to be aligned. The system is very flexible; we only need to access relevant CMM stripes so we can access subsets of data. The approach is a combination of the striping described above in section 4.1 and the CMM distribution described in section 4.2 with Hadoop orchestrating the search.

While the CMMs are being trained it is expedient to generate a MapReduce input file of input vectors to be used to produce the feature selections. These files will be split into batches by the MapReduce software and the results will be correlated to produce the feature selection scores. There is one input file per CMM stripe and the input vectors in each file represent the set of input vectors for recall to produce the feature selections.

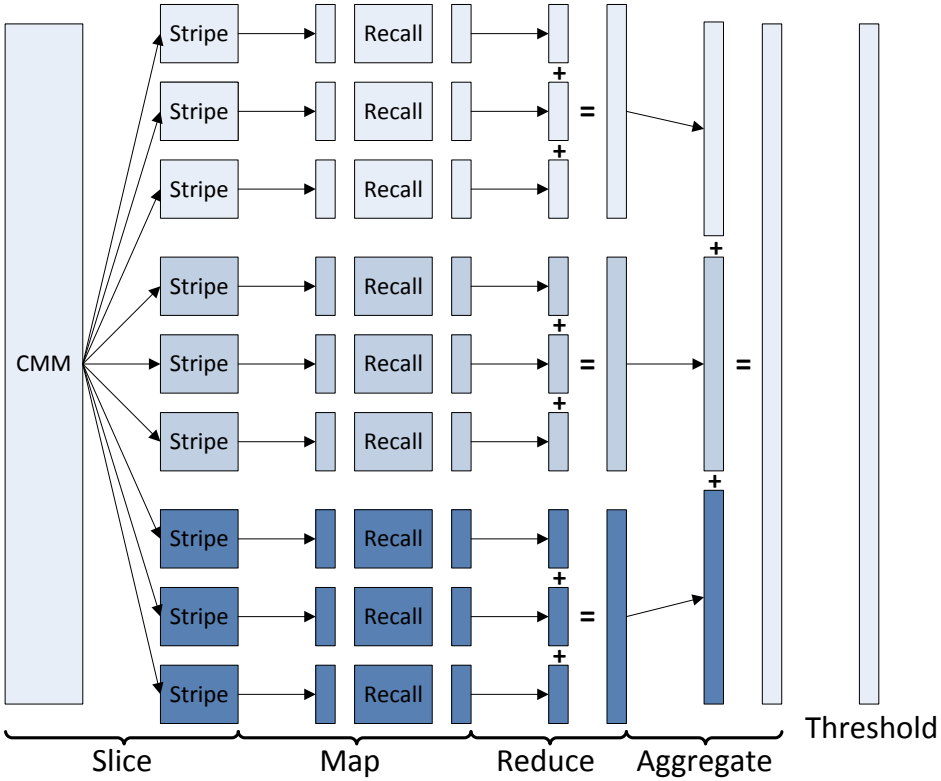

Figure 5 Figure showing distributed AURA recall in Hadoop. In the figure, there are three distributed compute nodes as shown by the shading with three CMM stripes per node (3 CPU cores per node and one stripe per core). Thus, the top three stripes are on one compute node spread across three cores. In the map phase, the required input vectors are applied to the CMM stripes and the summed output vector is recalled for each stripe. The summed output vector can be thresholded now or later following aggregation as described in section 4.3.1. During the reduce phase, these output vectors are aggregated at each compute node giving three aggregated vectors. Finally, the three vectors are combined.

Each CMM stripe that receives a search request, executes the recall process described in section 2.3. The candidate matches are the set of stored patterns that are close to the query in the feature space. In Hadoop the processing is coordinated by MapReduce (Shvachko et al., 2010). Hadoop YARN schedules the MapReduce tasks independently of the problem being solved. There is one Map job for each input file. Therefore, we model feature selection as a series of MapReduce jobs with each job representing one CMM stripe and the tasks are batches of file iterations (batch processing subsets of records) from the test data. The tasks are processed in parallel on distributed nodes. Each CMM stripe is read into a job. The recall function for CMM stripes is written as a Map task. Each MapReduce job invokes multiple Map tasks, each task represents a batch of recalls for a subset of input records, the batches execute in parallel. The Hadoop Mapper keeps track of the output vector versus record ID pairs so we know which output vector is associated with which record. The Reduce tasks perform the integer output vector thresholding as described in section 2.3 and write the data back into the file associated with the CMM stripe. Multiple feature selectors can be run in parallel, each executing as a series of MapReduce jobs. The CMMs for feature selection are immutable so subsequent iterations do not depend on the results (or changes) of the CMMs.

This whole MapReduce process has to be coordinated. If the MapReduce process is running at a single location then it can be coordinated as a Java class that initiates the individual jobs and then coordinates the results from all jobs to produce the feature selection scores. If the processing is geographically distributed then it needs a more complete coordinator. This can be achieved using for example the UNIX curl command and a monitor process that determines when curl has collected new data. Alternatively, it can be achieved using a distributed stream processor such as Apache Flume (https://flume.apache.org/) or Storm (https://storm.incubator.apache.org/). Essentially, whichever tool is used this is a three part process: initiate the feature selection process at each of the distributed nodes; retrieve the results data from the distributed nodes; and, monitor when the results have been returned from all nodes and combine them into a single unified result.

4.3.1 Stripe vectors

For Big data, the CMMs are too big to store in a one computer's memory. Hence, they need to be striped across multiple computers as in figure 4 and figure 5. Each CMM stripe returns a vector representing the matching results for the input vector with respect to that CMM stripe. Palm (2013) has extensively analysed representations in associative memories and found that sparse representations are optimal because the number of matrix operations is proportional to the number of set bits in the vectors. A sparse pattern will have fewest set bits and require fewest operations. For our feature selector, each CMM stripe can return its results as

1. an integer vector $\boldsymbol{S_k}$ (un-thresholded),
2. a thresholded vector $\boldsymbol{T_k}$ or
3. a list of the set bits in the thresholded vector.

Option 1 is the least efficient as, potentially, every column could have an integer score so the vector would be an integer vector of length $\boldsymbol{N}$ where $\boldsymbol{N}$ is the number of data records stored. This integer vector can be thresholded for option 2 which produces a binary vector. A binary vector requires less storage capacity than an integer vector (1 bit per element for the binary vector compared to 16 or 32 bits per element for the integer vector). For option 3, we would return a list of the set bits. For this we can exploit a compact list representation for representing binary vectors (Hodge & Austin, 2001). This compact list representation is similar to the pointer representation used in associative memories (Bentz, Hagstroem & Palm, 1997). It ensures that retrieval is proportional to the number of set bits in the thresholded output vector so is fast and scalable. The feature selection process produces a large set of output vectors from the CMM stripes; namely, all vectors necessary for all feature selectors. Option 3 allows AURA to be used for distributed processing with data sets of millions of records while using a relatively small amount of memory and with a massively reduced communication overhead. For example, if there were 10,000,000 records in the data set then a vector would have 10,000,000 elements. If only three records match (records; 8, 10 and 11) then processing {8,10,11} as indices requires much less time, memory and communication bandwidth compared to processing 10,000,000 binary digits. Hence, wherever possible we use option 3.

The results need to be amalgamated for each feature selector to produce the feature scores for that feature selector. The system maintains an index of what data are stored where and what each datum represents so the coordinating node can coordinate the matching, receive all matching data and determine the set of best matches across all searchable data. Each feature selector will have a separate amalgamate program running at the coordinating node. This program uses the required vectors and set bit counts returned from AURA to produce the feature score as described in sections 3 and 5.

5 Analysis of AURA Feature Selection

We demonstrate theoretically using a worked example that our framework vastly reduces the number of required computations compared to processing the feature selectors separately. The worked example provides an easy and simple illustration of the method on a small data size. We envisage using the feature selector on Big Data sets where Big Data refers to data sets that require at a minimum multiple CPUs but more likely multiple compute nodes to process in tractable time for the application. The larger the data set and the more time critical the data processing then the more important our computation

reduction will become. MI, CFS, CS, OR and GR can all use a single CMM representation for the data such as the CMM in Figure 6. This overall CMM is amenable to striping across the processing nodes to allow Hadoop processing in a similar fashion to Figure 4 and Figure 5. The framework is underpinned by Hadoop which has been thoroughly evaluated in the literature (Kumar et al., 2013). Hadoop is highly configurable large data set framework that can be optimised to the user's specific requirements, for example, optimising to minimise memory overhead, optimising for fastest processing or optimising to reduce communication overhead. Hence, we do not attempt to evaluate Hadoop itself here but just focus on how we minimise the number of feature selection computations to minimise processing. Users will use our framework to select the best feature selector for their data and application using their own specific criteria.

The feature selectors in section 3 have many commonalities when implemented in the unified AURA framework. We can demonstrate the commonalities by analysing 12 records from the Iris data set (Fisher, 1936). The Iris data are illustrated in Figure 6 (left) when trained into the CMM. The 12 records have been trained into a CMM using the four features and the class. Each feature is quantitative and has been subdivided into five quantisation bins of equal width. Figure 6 (right) shows the same data divided into four CMM stripes (***CMMStripe1, CMMStripe2, CMMStripe3*** and ***CMMStripe4***). The horizontal (row-based) striping means that the features "*sepal len"* and "*sepal width*" are in the top stripes and "*petal len*", "*petal width*" and the *class* are in the bottom two stripes. The vertical (column-based) striping means that the first 6 data records are stored in the left two stripes and the other 6 records in the right two stripes. If the data were time-series or sequential, the column-based striping would form two time frames with the oldest data in the left two stripes and the newest data in the right two stripes. The input vectors are stored in a file for each CMM or CMM stripe. These files can then be batch processed in the Hadoop framework described. Within the evaluation, we consider how the data and CMMs would be accommodated in our Hadoop framework.

MI, CFS, CS, OR and GR all use $\boldsymbol{BVf_i}$ (the binary vector where ($\boldsymbol{F_j}$=$\boldsymbol{f_i}$)), $\boldsymbol{BVb_i}$ (the binary vector representing the quantisation bin $\boldsymbol{bin(f_i)}$) and ***BVc*** (the binary vector representing all records that have class label ***c***). These only need to be extracted once and used in each feature selector as appropriate. For example in Figure 6, if we want all records where $\boldsymbol{1.12 \leq petal\ width < 1.58}$ then we activate row 17 of the CMM. We can then Willshaw threshold the resultant integer output vector ***S*** (0000**1111**0000) at level 1 and retrieve the binary thresholded vector ***T*** with a bit set for every matching record (bits 4,5,6,7). For the Hadoop distributed version, only the relevant CMM stripes are queried in Figure 6 (right). In this case, activating row 17 of ***CMMStripe3*** and ***CMMStripe4*** queries the relevant data. ***CMMStripe3*** will output thresholded vector $\boldsymbol{T_3}$ with bits 5 and 6 set and ***CMMStripe4*** will output $\boldsymbol{T_4}$ with bits 7 and 8 set. $\boldsymbol{T_3}$ and $\boldsymbol{T_4}$ can be concatenated to form a single vector thresholded vector ***T*** (as in figure 4) with bits 4, 5, 6 and 7 set. For the Hadoop distributed version, each CMM stripe ***CMMStripeX*** outputs a list of the indices of the set bits in $\boldsymbol{T_X}$ which are collected by the coordinator.

CFS, GR and MI all require $\boldsymbol{nBVf_i}$ a count of the number of data records where a particular feature has a particular value $\boldsymbol{F_j}$=$\boldsymbol{f_i}$ and ***BVc*** a count of the number of records where the class has a particular label ***C*=*c***. To count the number of records where $\boldsymbol{1.12 \leq petal\ width < 1.58}$, we retrieve the binary thresholded vector as above and count the number of set bits (bits 4, 5, 6 and 7 are set giving 4 matching records). For the Hadoop approach, we coordinate the retrieval as above, concatenate the lists to produce a single overall list of set bits and count the list length. $\boldsymbol{T_3}$ has bits 4 and 5 set and $\boldsymbol{T_4}$ has bits 6 and 7 set giving 4 matching records in total.

CFS, CS, OR, GR and MI all use ($\boldsymbol{BVf_i} \wedge \boldsymbol{BVc}$) and ($\boldsymbol{BVb_i} \wedge \boldsymbol{BVv}$) for qualitative and quantitative features respectively. For example, we can find all records where $\boldsymbol{4.6 \leq sepal\ len < 5.1}$ *and* the class is ***A*** by activating rows ***0*** and ***20*** of the CMM, thresholding ***S*** (1**222**000000) at Willshaw level 2 to give ***T*** with three bits set: column 1, 2 and 3 in Figure 6 (left). This takes more coordinating in the Hadoop framework as the data for the feature value may not be stored with the data for the class; they may be in different CMM stripes. In Figure 6 (right), activate row ***0*** in ***CMMStripe1*** and ***CMMStripe2*** and then activate row ***20*** in ***CMMStripe3*** and ***CMMStripe4***. The coordinating program needs to correlate the sections of the vector for the feature value and correlate the sections of the vector for the class to form a

single vector. ***CMMStripe1*** needs to be added (summed) with the output integer vector of ***CMMStripe3*** to give S_{1+3} and ***CMMStripe2*** needs to be added (summed) with the output integer vector of ***CMMStripe4*** to give S_{2+4}. The summed vectors can then be thresholded at 2 to give T_{1+3} with bits 1, 2 and 3 set (three matching records) and T_{2+4} with no bits set (no matches). The two thresholded output vectors are concatenated to produce T with bits 1, 2 and 3 set. If the thresholded vectors are stored as lists of indices (see section 4.3.1) then this is simply a task of finding the common indices between the two vectors.

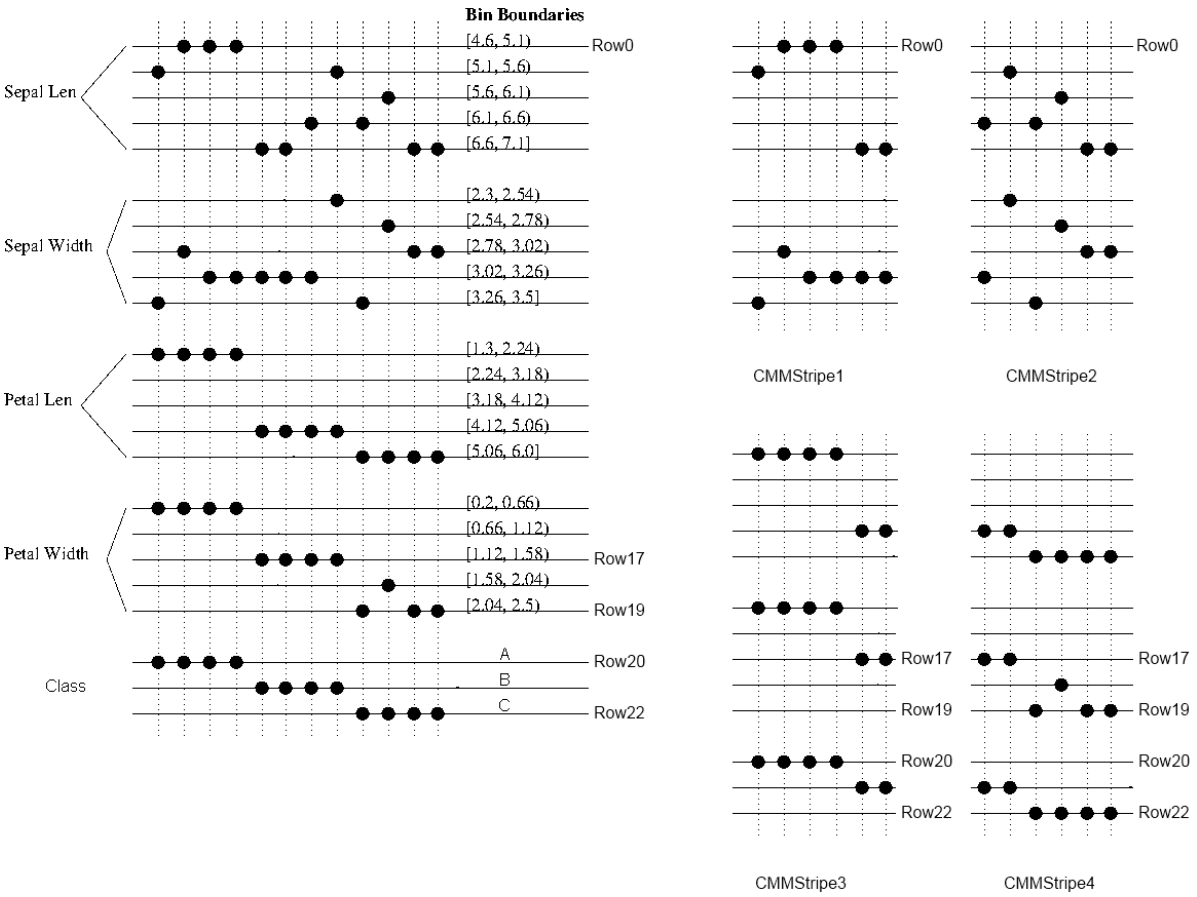

Figure 6 The 12 records from the iris data set, quantised and trained into a single AURA CMM (left) and subdivided across 4 stripes of the CMM (right). The letters in rows 20-22 indicate the class of the record: A=Iris-setosa, B=Iris-versicolor, C=Iris-virginica.

MI, CFS, CS, OR and GR all also need a count of the conjunction, that is $n(BVf_i \wedge BVc)$ and $n(BVb_i \wedge BVc)$ for qualitative and quantitative features respectively. Hence, we retrieve the binary thresholded vector T as above and count the set bits.

Rather than calculating these elements multiple times, we can take advantage of the commonalities by calculating each common value, binary vector or count only once and propagating the result to each feature selector that requires it. Following these common calculations, all necessary calculations will have been made for MI and GR. CFS just requires the pairwise feature versus feature analyses ($BVb_i \wedge BVb_k$). These are performed in the same way as the feature versus class analyses above. CS and OR require the manipulation of some of the binary vectors to produce the logical *or* vectors. This requires the coordination of the vectors. To find $(BVb_i) \vee (BV_c)$, we combine the list of set bits for (BVb_i) with the list of set bits for (BV_c) and count the resulting list length. By calculating the common elements first, the remainder of the calculations can be performed for each feature selector using either this CMM and processing the algorithms in series or by generating multiple copies of the CMM and processing them in parallel if sufficient processing capacity is available.

Once all of the binary vectors have been retrieved by the distributed Hadoop system, they need to be processed to calculate the feature scores as per section 3 using the various feature selectors. A coordinator program organises this in parallel. There is one feature score calculation process per feature selector (currently five feature selectors are described here).

For the Iris data set, there are 20 feature row activations $\boldsymbol{20 * BVb_i}$ and three class activations $\boldsymbol{3 * BVc}$. To calculate ($\boldsymbol{BVb_i \wedge BVc}$) requires 20 x 3 = 60 calculations. Hence, there are 83 common calculations (20+3+60) across all five feature selectors. CFS then needs to calculate ($\boldsymbol{BVb_i \wedge BVb_k}$) which would require 19! calculations if every feature value was compared to every other. However, CFS uses greedy forward search so that the number of comparisons is minimised (Hall, 1998) to a worst case of (20^2-20)/2=190. We have already extracted all $\boldsymbol{20 * BVb_i}$ binary vectors so CFS needs 190 logical *and*s but no CMM accesses. We have saved a minimum of 20 CMM accesses for $\boldsymbol{BVb_i}$ and a maximum of 190 CMM accesses for worst case forward search. Manipulating the binary vectors can be performed at the coordinating node and in parallel as a Hadoop batch process. CS requires the logical *or* vectors ($\boldsymbol{BVb_i \vee BVc}$). Again, we already have all $\boldsymbol{20 * BVb_i}$ binary vectors and all $\boldsymbol{3 * BVc}$ binary vectors so there are 20 x 3=60 logical *or*s to perform. Thus, we have saved a minimum of $\boldsymbol{20 * BV_{bi} + 3 * BV_c = 23}$ CMM accesses and potentially 60 CMM accesses if all 60 *or* operations were performed in the CMM. Thus MI requires 83 calculations, GR also requires 83, CFS requires 83 plus 190 and CS requires 83 plus 60. Without our reductions there would be 83+83+83+190+83+60 calculations. We have reduced this to 83+190+60. Additionally, 190+60 of these can use vectors already extracted so there is no need to access the CMM. We have saved 3 * 83=249 recalls from the CMM by finding common aspects, have removed a minimum of 20+23 further CMM recalls and have reduced the other calculations to logical operations on stored binary vectors. The minimum saving on CMM recalls is given by equation 16.

$$Saving = \Big(3 \times \Big(n(BVb_i) + n(BVc) + \big(n(BVb_i) \times n(BVc)\big)\Big)\Big) + \Big(\big(2 \times n(BVb_i)\big) + n(BVc)\Big) \quad (16)$$

6 Conclusion

Massive and complex data sources pose challenges for data mining but they also hold many opportunities. New information can be uncovered, vast timelines of data are available for analysis and the data models learned will be increasingly rich as the training data expands. How the data is represented needs to be carefully considered including careful preparation such as cleaning and selecting feature subsets. In this paper we have introduced a distributed processing framework for feature selection using the AURA neural network and Apache Hadoop. There are currently five feature selectors available which may be used independently or coupled with the AURA k-NN for classification or prediction.

All five feature selectors can use a single trained CMM. We have identified common aspects of the five feature selectors when they are implemented in the AURA framework and indicated how these common aspects may be processed as a common block. All remaining aspects of the feature selectors can then be implemented in parallel using duplicate copies of the trained CMM as compute resources allow. CMMs lend themselves to distributed processing as they can be striped (split) using both row-based and column-based striping. The CMM created for feature selection can be used directly for the AURA k-NN for classification or prediction and any unwanted features (those not selected by the feature selection) can simply be ignored (masked off). Alternatively, the CMM can be retrained with only the required data if processing speed and memory usage at recall time are the primary concern.

The AURA neural architecture has demonstrated superior training and recall speed compared to conventional indexing approaches such as hashing or inverted file lists (Hodge & Austin, 2001) and an AURA-based implementation of the MI feature selector was over 100 times faster than a standard implementation (Hodge, O'Keefe & Austin, 2006). This is further augmented by using the scalability of Hadoop. This combined platform allows rapid processing of feature selectors on large and high

dimensional data sets that cannot be processed on standard computers. We envisage using the method on data sets that require at a minimum multiple CPUs but more likely multiple compute nodes to process. The method is also best suited to data mining and analytics that processes a Big Data file in a longer term processing run such as overnight rather than on-line transaction processing which requires near real-time updating. The user can then evaluate the feature sets chosen by the feature selectors against their own data to determine the best feature selector and the best set of features. Additionally, each feature selector (MI, CFS, GR, CS and OR) generates scores for the features which can be used to weight the features during machine learning.

The technique is flexible and easily extended to other feature selection algorithms. By implementing a range of feature selectors in a single framework, we can also investigate ensemble feature selection where the results from a range of feature selectors are merged to generate a consensus overview of the best set of features to use.

We will investigate whether we can use Apache Spark, the in-memory data analytics and cluster computing framework (https://spark.apache.org/) to underpin the AURA feature selection framework. Apache Spark is closely coupled with Hadoop and allows YARN and MapReduce jobs to be run. Spark enables in-memory computing and is reputed to be up to 100 times faster than MapReduce (see https://spark.apache.org/). CMMs are optimised for in-memory processing so fit well with the Spark paradigm. A related development, Optimized Row Columnar (ORC) file format is currently being adopted by Spark. ORC is a file storage format that is tightly integrated with HDFS and provides optimizations for both read performance and data compression. An ORC file divides the data into groups of row data called stripes. This fits with the stripes used in AURA CMMs and would allow a direct mapping from ORC data file stripes to CMM stripes for optimised performance.

We plan to use the feature selection framework that we have developed in this paper in conjunction with the AURA k-NN for traffic analysis (Hodge, Jackson, & Austin, 2012; Hodge, Krishnan, Austin & Polak, 2010; Hodge, Krishnan, Austin & Polak, 2011), condition monitoring (Austin, Brewer, Jackson & Hodge, 2010) and railway infrastructure monitoring in the NEWTON Project (Hodge, O'Keefe, Weeks & Moulds, 2015).

7 Acknowledgements

This work was supported by UK Engineering and Physical Sciences Research Council (Grant EP/J012343/1).

8 References

Austin, J. (1995). Distributed associative memories for high speed symbolic reasoning. In R. Sun & F. Alexandre (Eds), *IJCAI '95 Working Notes of Workshop on Connectionist-Symbolic Integration: From Unified to Hybrid Approaches*, (pp. 87-93), Montreal, Quebec.

Austin, J., Brewer, G., Jackson T., & Hodge V. (2010). AURA-Alert: The use of binary associative memories for condition monitoring applications. In *Procs 7th Int'l Conf. on Condition Monitoring and Machinery Failure Prevention Technologies*, (pp. 699-711). Red Hook, USA: Curran Associates.

Bentz, H., Hagstroem M., & Palm G. (1997). Selection of relevant features and examples in machine learning. *Neural Networks*, *2*(4), 289 - 293.

Borthakur, D. (2008). HDFS architecture guide. HADOOP APACHE PROJECT http://pristinespringsangus.com/hadoop/docs/hdfs_design.pdf

Borthakur, D., Gray, J., Sarma, J. S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., ... & Aiyer, A. (2011). Apache Hadoop goes Realtime at Facebook. In *Procs 2011 ACM SIGMOD International Conference on Management of data* , (pp. 1071-1080). New York: ACM.

Casasent, D. & Telfer ,B. (1992). High capacity pattern recognition associative processors. *Neural Networks, 5*(4):251-261.

Chu, C., Kim, S., Lin, Y., Yu, Y., Bradski, G., Ng, A., & Olukotun, K. (2007). Map-reduce for machine learning on multicore. *Advances in Neural Information Processing Systems, 19*, pp. 281–288. MIT Press.

Dash, M., & Liu, H. (1997). Feature selection for classification. *Intelligent Data Analysis, 1*(3):131-156.

Ertöz, L., Steinbach, M., & Kumar, V. (2003) Finding clusters of different sizes, shapes, and densities in noisy, high dimensional data. In *Proc. 3rd SIAM Int'l Conf. on Data Mining*, (pp. 47-58).

Fayyad, U., & Irani, K. (1993). Multi-interval discretization of continuous-valued attributes for classification learning. In *Procs Int'l Joint Conf. on Artificial Intelligence*, (pp. 1022-1027). San Mateo, USA: Morgan Kaufmann.

Fisher, R. (1936). The use of multiple measurements in taxonomic problems. *Annual Eugenics, 7*(2), 179-188.

Fletcher, M., Jackson, T., Jessop, M., Liang, B., & Austin, J. (2006). The signal data explorer: A high performance grid based signal search tool for use in distributed diagnostic applications. In CCGrid 2006 - 6th IEEE Int'l Symp. on Cluster Computing and the Grid, (pp. 217-224), Singapore.

Forman, G. (2003). An extensive empirical study of feature selection metrics for text classification. *J. Mach. Learn. Res., 3*, 1289-1305.

Franks, B. (2012). *Taming the big data tidal wave: Finding opportunities in huge data streams with advanced analytics*. Hoboken, NJ: John Wiley & Sons.

Guyon, I., & Elisseeff, A. (2003). An introduction to variable and feature selection. *J. Mach. Learn. Res., 3*, 1157-1182.

Guyon, I., Weston, J., Barnhill S., & Vapnik, V. (2002). Gene selection for cancer classification using support vector machines. *Mach. Learn., 46*(1), 389-422.

Hall, M. (1998). Correlation-based Feature Subset Selection for Machine Learning. (Unpublished doctoral dissertation). University of Waikato, Hamilton, New Zealand.

Hall, M., & Holmes, G. (2003). Benchmarking attribute selection techniques for discrete class data mining. *IEEE Trans. on Knowl. & Data Eng., 15*(6), 1437-1447.

Hall, M., & Smith, L. (1997). Feature subset selection: a correlation based filter approach. In *Int'l Conf. on Neural Information Processing and Intelligent Information Systems*, (pp. 855-858). Berlin: Springer.

Hall, M., & Smith, L. (1998). Practical feature subset selection for machine learning. In *Procs of the 21st Australian Computer Science Conf.*, (pp. 181-191). Berlin: Springer.

Han, J., & Kamber, M. (2006). *Data mining: concepts and techniques*: *The Morgan Kaufmann Series in Data Management Systems*, Elsevier.

Hebb, D. (1949). *The organization of behavior: a neuropsychological theory*, New York: Wiley.

Hodge, V. (2011). *Outlier and Anomaly Detection: A Survey of Outlier and Anomaly Detection Methods*. LAP LAMBERT Academic Publishing.

Hodge, V., & Austin, J. (2001). An Evaluation of Standard Retrieval Algorithms and a Binary Neural Approach. *Neural Networks*, *14*(3).

Hodge, V., & Austin, J. (2005). A binary neural k-nearest neighbour technique. *Knowl. Inf. Syst., 8*(3):276-292, 2005.

Hodge, V., & Austin J. (2012). Discretisation of Data in a Binary Neural k-Nearest Neighbour Algorithm. *Tech. Report YCS-2012-473*, UK: University of York, Department of Computer Science.

Hodge, V., Jackson, T., & Austin, J. (2012). A binary neural network framework for attribute selection and prediction. In *4th Int'l Conf. on Neural Computation Theory and Applications*, Barcelona, Spain.

Hodge, V., Krishnan, R., Austin J., & Polak, J. (2010). A computationally efficient method for on-line identification of traffic control intervention measures. In *42nd Annual UTSG Conf*., University of Plymouth, UK.

Hodge, V., Krishnan, R., Austin J., & Polak, J. (2011). Short-term traffic prediction using a binary neural network. In *43rd Annual UTSG Conf*., Milton Keynes, UK.

Hodge, V., O'Keefe, S., & Austin, J. (2006). A binary neural decision table classifier. *NeuroComputing, 69*(16-18), 1850-1859.

Hodge, V., O'Keefe, S., Weeks, M., & Moulds, A. (2015). Wireless Sensor Networks for Condition Monitoring in the Railway Industry: A Survey, *IEEE Trans on Intelligent Transportation Systems, 16*(3), 1088–1106.

Jolliffe, I. (2002). *Principal Component Analysis*. (2nd Ed.). New York, USA: Springer-Verlag.

Kohavi, R., & John, G. (1997). Wrappers for feature subset selection. *Artif. Intell. J*., Special Issue on Relevance, *97*(1-2), 273-324.

Kumar V., et al. (2013). Apache Hadoop YARN: yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing* (SOCC '13). ACM, New York, NY, USA.

Laney, D. (2001). 3D Data Management: Controlling Data Volume, Velocity, and Variety. *Application Delivery Strategies* by META Group.

Liu, H., Motoda, H., Setiono, R., & Zhao, Z. (2010). Feature selection: An ever evolving frontier in data mining. *J. Mach. Learn. Res., 10*, 4-13.

Liu, H., & Setiono, R. (1995). Chi2: Feature selection and discretization of numeric attributes. In *Procs IEEE 7th Int'l Conf. on Tools with Artificial Intelligence*, (pp. 338-391). IEEE.

Liu, H., & Setiono, R. (1996). A Probabilistic Approach to Feature Selection - A Filter Solution. In *Procs of 13th Int'l Conf. on Machine Learning* (pp. 319-327). San Mateo, USA: Morgan Kaufmann.

McCallum, A., Nigam, K., & Ungar, L. (2000). Efficient clustering of high-dimensional data sets with application to reference matching. In *Procs 6th ACM SIGKDD Int'l Conf. on Knowledge Discovery & Data Mining* (pp. 169-178). New York: ACM.

Palm, G. (2013). Neural associative memories and sparse coding, *Neural Networks, 37*, 165-171.

Quinlan, J. (1986). Induction of decision trees. *Mach. Learn., 1*, 81-106.

Quinlan, J. (1992). *C4.5 Programs for Machine Learning*. San Mateo, USA: Morgan Kaufmann.

Reggiani, C. (2013). Scaling feature selection algorithms using MapReduce on Apache Hadoop. (Master's thesis, Politecnico di Milano, Italy). Retrieved from https://www.politesi.polimi.it/bitstream/10589/81201/1/2013_07_Reggiani.pdf.

Rutman, N. (2011). Map/reduce on lustre, (white paper). Technical report, Havant, UK: Xyratex Technology Limited.

Shvachko, K., Hairong, K., Radia S., & Chansler, R. (2010). The Hadoop distributed file store system. In *Procs IEEE 26th Symp. on Mass Storage Systems and Technologies* (pp. 1-10). IEEE.

Singh, S., Kubica, J., Larsen, S., & Sorokina, D. (2009). Parallel Large Scale Feature Selection for Logistic Regression. In, SIAM International Conference on Data Mining (SDM) (pp. 1172-1183).

Sun, Z. (2014). Parallel feature selection based on MapReduce. In, Computer Engineering and Networking Lecture Notes in Electrical Engineering, Vol. 277, (pp. 299-306). Springer.

Varela, P., Martins, A., Aguiar, P., & Figueiredo, M. (2013). An Empirical Study of Feature Selection for Sentiment Analysis. In, *9th Conference on Telecommunications*, Conftele 2013, Castelo Branco.

Weeks, M., Hodge, V., & Austin, J. (2002). A hardware accelerated novel IR system. In *Procs 10th Euromicro Workshop (PDP-2002)*. IEEE Computer Society.

Weeks, M., Hodge, V., O'Keefe, S., Austin, J., & Lees, K. (2003). Improved AURA k-nearest neighbour approach. In *Artificial Neural Nets Problem Solving Methods* (pp. 663-670). Berlin: Springer.

Wettscherek, D. (1994). A study of distance-based machine learning algorithms. (Unpublished doctoral dissertation). Oregon State University, Corvallis, USA.

Willshaw, D., Buneman O., & Longuet-Higgins, H. (1969). Non-holographic associative memory. *Nature, 222*, 960-962.

Witten, I., & Frank, E. (2000). *Data Mining: practical machine learning tools and techniques with Java implementations*. San Mateo, USA: Morgan Kaufmann.

Yang, Y., & Pedersen, J. (1997). A comparative study on feature selection in text categorization. In *Procs 14th Int'l Conf. on Machine Learning* (pp. 412-420).

Zhang, C., Kumar, A., & Ré, C. (2014). Materialization optimizations for feature selection workloads. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (pp. 265-276). ACM.

Zikopoulos, P., & Eaton, C. (2011). *Understanding big data: Analytics for enterprise class Hadoop and streaming data*. New York: McGraw-Hill.