University of Sydney

Final year thesis

# Implementation of Random Projection Algorithms Using an FPGA

*Author:*
Joshua Spence

*Supervisor:*
Philip Leong

A thesis submitted in fulfilment of the requirements for the degree of Bachelor of Engineering (Computer) in the School of Electrical Engineering at The University of Sydney

2012

# Acknowledgements

This project could not have been completed without the guidance of my project supervisor, *Philip Leong*. He always made himself available to assist me with the understanding of many new and unknown concepts. His expertise in the field was a great asset and greatly contributed to my own understanding of the ideas explored in this thesis. His support and guidance was invaluable, and were instrumental to the success of my project.

Much of the works completed during this project could not have been performed without the prior research of *Sanjay Chawla* and *Nguyen Lu Dang Khoa*. Both were generous enough to lend their own support and expertise in order to gain an understanding of the implemented random projection algorithms.

Additionally, I must thank *Michael Frechtling* for his assistance and advice throughout the project. He assisted me with Xilnix tools such as "AutoESL", and also offered me valuable advice regarding my future professional career.

# Publications

This thesis is based on the following publications:

1. Timothy de Vries, Sanjay Chawla and Michael Houle. "Finding Local Anomalies in Very High Dimensional Space". In: *10th IEEE International Conference on Data Mining*. 2010, pp. 128–137. DOI: 10.1109/ICDM.2010.151

2. Nguyen Lu Dang Khoa. "Large Scale Anomaly Detection and Clustering Using Random Walks". PhD thesis. University of Sydney, Mar. 2012

3. Stephen D. Bay and Mark Schwabacher. "Mining distance-based outliers in near linear time with randomization and a simple pruning rule". In: *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. KDD 2003. Washington, D.C.: ACM, Aug. 2003, pp. 29–38. ISBN: 1-58113-737-0. DOI: 10.1145/956750.956758. URL: http://doi.acm.org/10.1145/956750.956758 (visited on 21/05/2012)

**Abstract**

The prediction of the stock market has become an issue of great interest in the areas of finance, mathematics and engineering; due mainly to the great potential financial gain. Researchers have devised various algorithms and differing approaches to the problem of stock market analysis, with varying degrees of success.

A major outstanding issue for stock market analysis is the effective and efficient detection of local anomalies in the input data sets, which are inherently highly multidimensional. Many naïve algorithms are highly inefficient and others fail to adequately detect local anomalies altogether. It had become a time-vs-correctness trade-off in which no acceptable compromise could be reached.

However, researchers are starting to explore the relatively new concept of applying "random projections" to the highly multidimensional data sets. Research has suggested that by applying these random projections, they are able to significantly reduce the dimensionality (and consequently the computational complexity) of the data sets, whilst sufficiently retaining the inherent properties of that data set — at least so much so as anomaly detection is concerned.

Anomaly detection is important because it allows otherwise-accurate machine learning algorithms such as neural networks to more accurately model and predict the stock exchange data by ignoring anomalous data, which likely doesn't effect the state of the model to any significant degree.

*Sanjay Chawla* from the University of Sydney has in recent years conducted and supervised new and exciting research oriented around random projections. In particular, *Nguyen Lu Dang Khoa*, under the supervision of *Sanjay Chawla* evaluated the use of traditional distance metrics, such as Euclidean distance and Mahalanobis distance, in the application of local anomaly detection. *Nguyen Lu Dang Khoa* proposed the use of the 'commute time' metric, derived from random walks on graphs, in anomaly detection.

# Contents

# List of Figures

# List of Algorithms

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

Anomaly detection is an important technique which can be applied to a wide range of applications. There are many techniques to detect and measure anomalies, each usually most applicable to some specific problem domain. A general algorithm for anomaly detection has proved difficult to design, due to various challenges associated with defining and measuring anomalies. Most existing approaches are based on statistical or geometrical measures involving distance. Whilst many algorithms work well for some subset of input data sets, it has been difficult to discover an algorithm which performs both correctly and efficiently on all possible data sets.

Previous research has found merit in applying randomization techniques to highly multivariate data sets in order to reduce the dimensionality of these data sets whilst maintaining their fundamental and statistical properties. Such reduction of the dimensionality of data, assuming it can be performed efficiently, allows previously-unscalable anomaly detection algorithms to be practically applied to a wider range of data and applications.

Anomaly detection is an important and contemporary problem in the field of computer science, and is of particular interesting to stock market analysis, network intrusion detection and image comparison.

## 1.2 Contributions of this thesis

One key difficulty in anomaly detection is the efficient scaling of a general algorithm to apply to highly multivariate data. In this thesis, I explore the use of randomization techniques (such as random projections and commute time) in anomaly detection algorithms. These techniques provide encouraging results with regards to the run-time complexity of an algorithm.

Furthermore, in this thesis I attempt to make observations and analysis of the run-time of anomaly detection using randomization techniques, so as to identify steps in the algorithms that bottleneck the algorithm's performance. Through this identification it would be possible to improve the *actual* run-time performance of the algorithm by utilisation the advantages of reconfigurable computing.

## 1.3 Organization

The rest of this thesis is organized as follows. In chapter 2, we provide background to various anomaly detection techniques, as well as randomization techniques. In order to provide the reader with an understanding of the background topics, a brief background is given to various topics in linear algebra, vector calculus and graph theory. In chapter 3, we provide an overview of reconfigurable computing, including an explanation of Field-Programmable Gate Arrays (FPGAs).

In chapter 4, we profile the execution of an anomaly detection algorithm and explore possible improvements to the algorithm, in particular by outsourcing various stages of the algorithm to an FPGA device. In chapter 5, we describe the implementation of the improved algorithm and detail the process that was followed in order to construct the hardware processing device. In chapter 6, we record results obtained by benchmarking the device that was previously designed and constructed, and comparing the expected improvements to the algorithm's execution with the measured results.

We conclude in chapter 7 by reflecting upon the results obtained through this research, and making suggestions for further research in this topic.

## 1.4 Schedule

A Gantt chart showing the anticipated schedule for the project is shown in Figure 1.1. This will be updated as the project progresses.

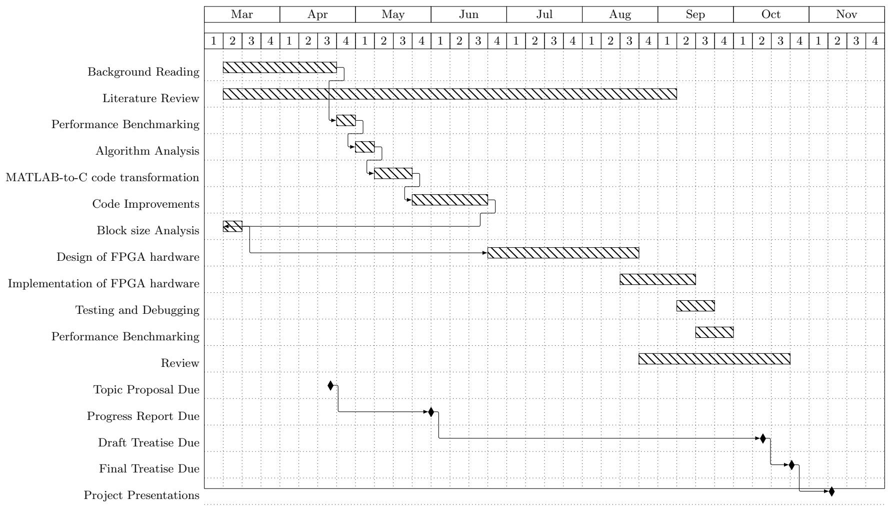



**Figure 1.1:** Schedule for thesis work

# Chapter 2

# Background

## 2.1 Anomaly detection

Anomaly detection is the process of detecting patterns in a given data set that do not conform to an "expected" behavior [9], although it is often difficult to accurate predict expected patterns and distributions for data sets. The terms 'anomaly' and 'outlier' are used synonymously, both within this thesis and more generally in the field of statistics.

According to Hawkins [19]:

> An outlier is an observation which deviates so much from the other observations as to arouse suspicions that it was generated by a different mechanism.

Anomaly and outlier detection are challenging areas that have gained much interest within the field of computer science. The importance of anomaly detection is due to the fact that anomalies in data translate to significant (and often critical) actionable information in a wide variety of application domains [9]. Over time, many techniques for anomaly detection have been developed for specific application domains, as well as more generic techniques [9].

### 2.1.1 What are anomalies?

Anomalies are patterns in data that do not conform to a well defined notion of normal behavior. Figure 2.1 illustrates anomalies in a simple 2-dimensional data set. The data has two normal regions, $N_1$ and $N_2$, since most observations lie in these two regions. Points that are sufficiently far away from the regions, such as points $o_1$ and $o_2$, and points in region $O_3$, are considered to be anomalies.

### 2.1.2 Challenges

A straightforward anomaly detection approach, is to define a region representing 'normal' behaviour and declare any observation in the data which does not belong to this normal region as an anomaly. But several factors make this apparently simple approach very challenging:

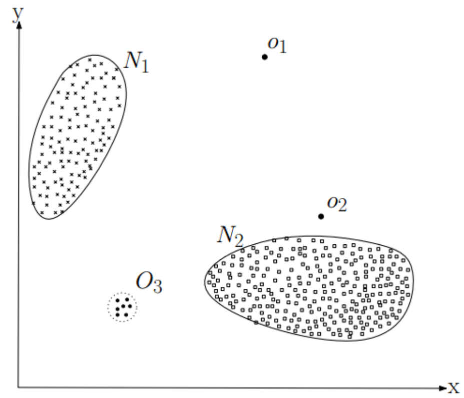



**Figure 2.1:** A simple example of anomalies in a 2-dimensional data set [9].

- Defining a normal region which encompasses every possible normal behaviour is very difficult. In addition, the boundary between normal and anomalous behaviour is often not precise. Thus an anomalous observation which lies close to the boundary can actually be normal, and vice-versa.
- When anomalies are the result of malicious actions, the malicious adversaries often adapt themselves to make the anomalous observations appear like normal, thereby making the task of defining normal behavior more difficult.
- In many domains normal behavior keeps evolving and a current notion of normal behavior might not be sufficiently representative in the future.
- The exact notion of an anomaly is different for different application domains. For example, in the medical domain a small deviation from normal (for example, fluctuations in body temperature) might be an anomaly, while similar deviation in the stock market domain (for example, fluctuations in the value of a stock) might be considered as normal. Thus applying a technique developed in one domain to another is not straightforward.
- Availability of labeled data for training/validation of models used by anomaly detection techniques is usually a major issue.
- Often the data contains noise which tends to be similar to the actual anomalies and hence is difficult to distinguish and remove.

Due to the above challenges, the anomaly detection problem, in its most general form, is not easy to solve. In fact, most of the existing anomaly detection techniques solve a specific formulation of the problem. The formulation is induced by various factors such as nature of the data, availability of labeled data, type of anomalies to be detected, etc. Often, these factors are determined by the application domain in which the anomalies need to be detected.

Researchers have adopted concepts from diverse disciplines such as statistics, data mining, statistics, information theory and spectral theory in order to gain an increased understanding of anomalies [9].

### 2.1.3 Similar problems

Anomaly detection is an intentionally broad specifier for a class of more-specific statisctical challenges. For example, whilst being distinct from, anomaly detection is a similar problem (in terms of complexity and approach) to that of *noise removal* and *noise accomodation*, both of which are aimed at removing the effects of unwanted *noise* in the data. Noise can be defined as any data which is not of specific interest to the analyst, but in its presence hinders data analysis techniques [9]. It is often critical to data analysis to remove or mitigate the effects that noise has to the properties of the host data set.

In contrast, the problem of *novelty detection* can often be impeded by techniques that attempt to remove anomalous data from a data set. *Novelty detection* is process of discovering emerging patterns in a data set, to provide an indication of the future state of a system. The distinction between novel pattern and anomalies is that novel patterns are incorporated into the data model after detection [9].

### 2.1.4 Classification

In general, two different kinds of outliers exist: global outliers and local outliers. Global outliers are distinct with respect to the whole data set, while local outliers are distinct with respect to data points in their local neighbourhood [43]. The task of global outlier detection has undergone much research [citation needed], but this has not been the case for local outlier detection. In the paper "Density-preserving projections for large-scale local anomaly detection", Vries, Chawla and Houle optimise the use of local outlier factor (LOF) for large and high-dimensional data and propose projection-indexed nearest-neighbours (PINN) — a novel technique that exploits extended nearest-neighbour sets in a reduced-dimensional space — to create an accurate approximation for k-nearest-neighbour distances, which is used as the core density measurement within LOF [43].

### 2.1.5 Types of anomalies

Anomalies can be classified into three categories [9]:

**Point anomaly** If an individual data instance can be considered as anomalous with respect to the rest of data, then the instance is termed as a point anomaly. This is the simplest type of anomaly. Referring to Figure 2.1, points $o_1$ and $o_2$, as well as all points in region $O_3$ lie outside the boundary of the normal regions, and are hence point anomalies.

**Contextual anomalies** If a data instance is anomalous in a certain context, but not otherwise, then it is termed a contextual anomaly. The notion of a context is induced by the structure in the data set and has to be specified as part of the problem formulation.

Contextual anomalies have been most commonly explored in time-series data and spatial data. Figure 2.2 shows one such example for a temperature time series which shows the monthly temperature of an area over last few years. A temperature of $35°F$ might be normal during the winter

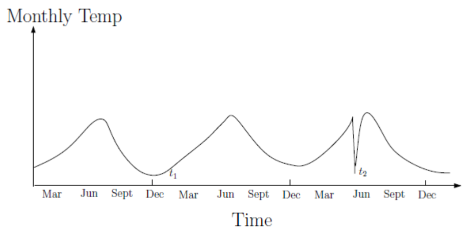



**Figure 2.2:** Contextual anomaly $t_2$ in a temperature time series. Note that the temperature at time $t_1$ is same as that at time $t_2$ but occurs in a different context and hence is not considered as an anomaly [9].

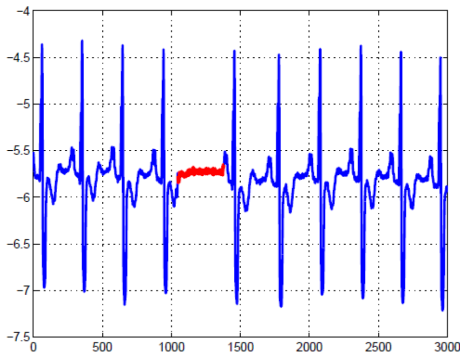



**Figure 2.3:** Collective anomaly corresponding to an *Atrial Premature Contraction* in an human electrocardiogram output [15].

(at time $t_1$) at that place, but the same value during summer (at time $t_2$) would be an anomaly.

**Collective anomalies** If a collection of related data instances is anomalous with respect to the entire data set, it is termed as a collective anomaly. The individual data instances in a collective anomaly may not be anomalies by themselves, but their occurrence together as a collection is anomalous. Figure 2.3 illustrates an example which shows a human electrocardiogram output. The highlighted region denotes an anomaly because the same low value exists for an abnormally long time. Note that that low value by itself is not an anomaly.

### 2.1.6 Approaches

**Classical** A point is declared to be an outlier if its distance from the mean is sufficiently large.

**Principal Component Analysis** An outlier is usually declared if the point is sufficiently far away from the subspace spanned by the eigenvectors corresponding to the highest eigenvalues.

**Distance based** A point can be declared to be an outlier if its distance to its kth nearest-neighbour is sufficiently large.

**Statistical based** Statistical methods are often model-based and assume that the data should follow some distribution. With knowledge of the distribution, data point are evaluated by their fitness to the assumed distribution. If the probability of a data instance is less than a certain threshold, then that data point is considered an anomaly.

Although distance is an effective non-parametric approach to detecting outliers, the drawback is the amount of computation time required. Straightforward algorithms, such as those based on nested loops, typically require $O(N^2)$ distance computations. This quadratic scaling means that it will be very difficult to mine outliers as we tackle increasingly larger data sets. This is a major problem for many real databases where there are often millions of records [7].

#### Distance based

In this approach, one looks at the local neighborhood of points for an example typically defined by the $k$ nearest examples (also known as neighbours). If the neighbouring points are relatively close, then the example is considered normal; if the neighbouring points are far away, then the example is considered unusual. The advantages of distance-based outliers are that no explicit distribution needs to be defined to determine unusualness, and that it can be applied to any feature space for which we can define a distance measure [7].

Researchers have tried a variety of approaches to find these outliers efficiently. The simplest are those using nested loops [7]. In the basic version one compares each example with every other example to determine its $k$ nearest neighbors. Given the neighbors for each example in the data set, simply select the top $n$ candidates according to the outlier definition. This approach has quadratic complexity as we must make all pairwise distance computations between examples.

Another method for finding outliers is to use a spatial indexing structure such as a KD-tree, R-tree, or X-tree to find the nearest neighbors of each candidate point. One queries the index structure for the closest $k$ points to each example, and as before one simply selects the top candidates according to the outlier definition. For low-dimensional data sets this approach can work extremely well and potentially scales as $O(N \log N)$ if the index tree can find an example's nearest neighbors in $\log N$ time. However, index structures break down as the dimensionality increases [7].

#### Statistical based

A common distribution considered when modelling data is the 'Normal' distribution. Using this model, the probability that a data instance lies within $k$ standard deviations $\sigma$ from the mean $\mu$ is the area between $\mu - k\sigma$ and $\mu + k\sigma$.

### 2.1.7 Local Outlier Factor

'Local Outlier Factor' is a formula that captures the degree to which a data point is an outlier with respect to its local neighbourhood. In this context, 'local' means that the determination of the data points does not depend on knowledge of the global distribution of the data set.

## 2.2 Vectors and Matrices

### 2.2.1 Eigenvectors and Eigenvalues

This section will briefly recall some basic definitions of eigenvectors and eigenvalues, as well as some of their basic properties.

A vector $\mathbf{v}$ is an eigenvector of a matrix $M$ of eigenvalue $\lambda$ if:

$$M\mathbf{v} = \lambda\mathbf{v} \tag{2.1}$$

If $\mathbf{v_1}$ is an eigenvector of $M$ of eigenvalue $\lambda_1$, $\mathbf{v_2}$ is an eigenvector of $M$ of eigenvalue $\lambda_2 \neq \lambda_1$, and $M$ is symmetric, then $\mathbf{v_1}$ is orthogonal to $\mathbf{v_2}$.

For a symmetric matrix $M$, the multiplicity of an eigenvalue $\lambda$ is the dimension of the space of eigenvectors of eigenvalue $\lambda$. Also recall that every $n\times n$ symmetric matrix has $n$ eigenvalues, counted with multiplicity. Thus, it has an orthonormal basis of eigenvectors, $\{\mathbf{v_1} \quad \ldots \quad \mathbf{v_n}\}$ with eigenvalues $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$ so that:

$$M\mathbf{v_i} = \lambda_i\mathbf{v_i} \quad \forall i \tag{2.2}$$

If we let $V$ be the matrix whose $i$th column is $v_i$ and $\Lambda$ be the diagonal matrix whose $i$th diagonal is $\lambda_i$, we can write this more compactly as:

$$MV = V\Lambda \tag{2.3}$$

Multiplying by $V^T$ on the right, we obtain the eigen-decomposition of $M$:

$$M = MVV^T = V\Lambda V^T = \sum_i \lambda_i \mathbf{v_i}\mathbf{v_i^T} \tag{2.4}$$

### 2.2.2 Eigen decomposition

### 2.2.3 Laplacian Matrices

Recall that a weighted, undirected graph $G = (V, E, w)$ is essentially an undirected graph $G = (V, E)$ along with a function $w : E \to \Re^+$, where $\Re^+$ denotes the set of positive real numbers.

The adjacency matrix of a weighted graph $G$ is be denoted $A_G$, and is given by:

$$A_G(i,j) := \begin{cases} w(i,j) & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

The degree matrix of a weighted graph $G$, denoted $D_G$, is the diagonal matrix such that:

$$D_G(i,i) = \sum_j A_G(i,j) \tag{2.6}$$

A Laplacian Matrix is a matrix representation of a graph, defined by the equation:

$$L_G = D_G - A_G \tag{2.7}$$

For a vector $\mathbf{x} \in \Re^V$, the Laplacian quadratic form of $G$ is:

$$\mathbf{x}^T L\mathbf{x} = \sum_{(u,v)\in E} w_{u,v}(\mathbf{x}(u) - \mathbf{x}(v))^2 \tag{2.8}$$

From Equation 2.8, it can be seen that $L$ provides a measure of the smoothness of $\mathbf{x}$ over the edges in $G$. The more $\mathbf{x}$ jumps over an edge, the larger the quadratic form becomes.

It is often more convenient to consider the normalized Laplacian of a graph instead of the Laplacian [37]. The normalized Laplacian is given by $D^{-1/2}LD^{-1/2}$ and is more closely related to the behaviour of random walks.

Now, let $G_{1,2}$ be a graph on two vertices with a single edge of weight 1.

$$L_{G_{1,2}} := \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \tag{2.9}$$

For the graph with $n$ vertices and just one edge between vertices $u$ and $v$, we can define the Laplacian Matrix similarly. For concreteness, I'll call this graph $G_{u,v}$. It's Laplacian Matrix is the $n \times n$ matrix whose only non-zero entries are in the intersections of rows and columns $u$ and $v$. The $2 \times 2$ matrix at the intersections of these rows and columns is, of course:

$$\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \tag{2.10}$$

For a weighted graph $G = (V, E, w)$, we define:

$$L_G := \sum_{(u,v) \in E} w(u,v) L_{G_{u,v}} \tag{2.11}$$

### Properties

Laplacian matrices of graphs are symmetric, have zero row-sums, and have non-positive off-diagonal entries. We call any matrix that satisfies these properties a Laplacian matrix, as there always exists some graph for which it is the Laplacian [37].

For a graph $G$ and its Laplacian Matrix $L$ with eigenvalues $\lambda_0 < \lambda_1 < \ldots < \lambda_{n-1}$:

- $L$ is a symmetric matrix. This means the eigenvalues of $L$ are real, and its eigenvectors are real and orthogonal.
- $L$ is always positive-semidefinite ($\forall i, \lambda_i \geq 0; \lambda_0 = 0$).
- Let $G = (V, E)$ be a graph, and let $0 = \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$ be the eigenvalues of its Laplacian Matrix. Then, $\lambda_2 > 0$ if and only if $G$ is connected.
- The number of times 0 appears as an eigenvalue in the Laplacian Matrix is the number of connected components in the graph.
- $\lambda_0$ is always 0 because every Laplacian Matrix has an eigenvector of $\begin{bmatrix} 1 & 1 & \ldots & 1 \end{bmatrix}$ that, for each row, adds the corresponding node's degree to a "-1" for each neighbour, thereby producing zero by definition.
- The smallest non-zero eigenvalue of $L$ is called the spectral gap.

- If we arbitrarily assign an orientation to the edges in $G$ and label each edge, then we can define the vertex edge incidence matrix $Q$ by:

$$Q_{ij} := \begin{cases} 1 & \text{if } e_j \text{ starts from } i \\ -1 & \text{if } e_j \text{ ends at } i \\ 0 & \text{otherwise} \end{cases} \tag{2.12}$$

  Then the Laplacian Matrix $L$ satisfies $L = Q^T Q$, regardless of the orientation of the edges.

- The second smallest eigenvalue of $L$ ($\lambda_2$) is the algebraic connectivity of $G$. $\lambda_2 > 0$ if and only if $G$ is connected.

**Applications**

An interesting application of Laplacian matrices is that of modelling electrical flow in a network resistors. In this model, the vertices of the graph correspond to points at which current can be added to or removed from the circuit. Edges in the graph correspond to resistors, with the edge weight equal to the conductance of the electrical resistor.

If $\mathbf{p} \in \Re^V$ denotes the vector of potentials and $\mathbf{i}_{ext} \in \Re^V$ the vectors of currents entering and leaving vertices, then these satisfy the relation:

$$L\mathbf{p} = \mathbf{i}_{ext} \tag{2.13}$$

This equation can be exploited to o compute the effective resistance between pairs of vertices [37]. The effective resistance between vertices $u$ and $v$ is the difference in potential one must impose between $u$ and $v$ to flow one unit of current from $u$ to $v$. To measure this, we compute the vector $\mathbf{p}$ for which $L\mathbf{p} = \mathbf{i}_{ext}$, where:

$$\mathbf{i}_{ext}(x) = \begin{cases} 1 & \text{for } x = u \\ -1 & \text{for } x = v \\ 0 & \text{otherwise} \end{cases} \tag{2.14}$$

We then measure the difference between $\mathbf{p}(u)$ and $\mathbf{p}$(v).

## 2.3 Commute Time

### 2.3.1 Introduction

Commute time is a robust distance metric derived from a random walk on graphs [23]. In “Large Scale Anomaly Detection and Clustering Using Random Walks”, Khoa demonstrated how commute time can be used as a distance measure for data mining tasks such as anomaly detection and clustering. A prohibitive limitation of this technique is that the calculation of commute time involves the eigen decomposition of the graph Laplacian, making it impractical for large graphs.

A major advantage of using commute time as a distance metric for outlier detection is that it effectively captures not only the distances between data points but also the density of the data [citation needed]. This property results in a distance metric that can be effectively used to capture global, local and group anomalies.

The commute time between two nodes $i$ and $j$ in a graph is the number of steps that a random walk, starting from $i$ will take to visit $j$ and then come back to $i$ for the first time. The fact that the commute time is averaged over all paths (and not just the shortest path) makes it more robust to data perturbations and it can also capture graph density [23]. Since it is a measure which can capture the geometrical structure of the data and is robust to noise, commute time can be applied in methods where Euclidean or other distances are used and thus the limitations of these metrics can be avoided.

### 2.3.2 Limitations

The computation of commute time requires the eigen decomposition (see subsection 2.2.2) of the graph Laplacian matrix (see subsection 2.2.3), a computation which takes $O(n^3)$ time and thus is not practical for large graphs [citation needed]. Methods to approximate the commute time to reduce the computational time are required in order to efficiently use commute time in large datasets.

### 2.3.3 Anomaly Detection Using Commute Time

## 2.4 Distance metrics

Distance is a quantitative description of how far apart two objects are. Mathematically, a distance or metric is a function describing how close or far away data points in some space are from each other [23].

### 2.4.1 Euclidean distance

An Euclidean distance between two data points in a space is the norm of the difference between two vectors corresponding to these data points [23]. Euclidean distance is extremely sensitive to the scale of the features involved. When dealing with features of vastly different scales, the effects of the larger feature dominant over the smaller feature in terms of the Euclidean distance. This problem is usually solved by normalizing the data values. Another issue, however, with Euclidean distance is that it is unable to take into account any correlation between data features.

### 2.4.2 Mahalanobis distance

Mahalanobis distance is a distance measure that considers the covariance between data features. Mahalanobis distance, however, is extremely sensitive to anomalies as anomalies affect both the mean and the covariance of the data.

### 2.4.3 Graph geodesic distance

## 2.5 Markov chains

A 'Markov chain' is a chance process in which the outcome of a given experiment can affect the outcome of the next experiment [17]. For a Markov chain, we have a set of states $S = \{s_1, s_2, \ldots, s_r\}$ with a process starting in one of the states and moving from state $s_i$ to $s_j$ with a probability $p_{ij}$ not dependent upon which states the chain was in before the current state. The probabilities $p_{ij}$ are called *transition probabilities*, and the complete matrix $\mathbf{P}$ of probabilities is known as the *transition matrix*.

The probability that, given the chain is in state $i$ now, it will be in state $j$ in two steps is denoted by $p_{ij}^{(2)}$. In general, if a Markov chain has $r$ states, then:

$$p_{ij}^{(2)} = \sum_{k=1}^{r} p_{ik} pkj \tag{2.15}$$

## 2.6 Random projections

## 2.7 Random walks on graphs

Assume we are given a connected undirected and weighted graph $G = (V, E, W)$ with edge weights $(w_{ij})_{i,j \in V} >= 0$ be the graph adjacency matrix. A degree of a node $i$ is $d_i = \sum_{j \in N(i)} w_{ij}$ where $N(i)$ is a set of neighbours of node $i$. All nodes nonadjacent to $i$ are assumed to have a weight of $w_{ij} = 0$.

A random walk is a sequence of nodes on a graph visited by a random walker: starting from a node, the random walker moves to one of its neighbours with some probability. Then from that node, it proceeds to one of its own neighbours with some probability, and so on [23]. The random walk is a finite Markov chain that is time-reversible, which means the reverse Markov chain has the same transition probability matrix as the original Markov chain [29].

The probability that a random walker selects a particular node from is neighbours is determined by the edge weights of the graph. The larger the weight $w_i j$ of the edge connecting nodes $i$ and $j$, the more often the random walker travels through that edge.

## 2.8 Nearest Neighbour Algorithms

## 2.9 Solvers

### 2.9.1 Spielman-Teng Solver

Spielman and Teng presented a randomised algorithm that, on input a symmetric, weakly diagonally dominant $n \times x$ matrix $A$ with $m$ non-zero entries and an $n$-vector $\mathbf{b}$, produces an $\tilde{\mathbf{x}}$ such that $\left\| \tilde{\mathbf{x}} - A^{\dagger}\mathbf{b} \right\|_A \leq \epsilon \left\| A^{\dagger}\mathbf{b} \right\|_A$ in expected time:

$$m \log^{O(1)} n \log(1/\epsilon) \tag{2.16}$$

# Chapter 3

# Reconfigurable Computing

## 3.1 Introduction

In the computer and electronics world, computations are performed either in hardware or in software. Computer hardware provides highly optimized resources for quickly performing critical tasks, but is permanently configured to a single task or application. Computer software offers a flexible approach, but is orders of magnitude worse than a hardware implementation in terms of performance, silicon area efficiency and power consumption.

FPGAs are devices that combine the advantages of hardware implementations with the flexibility of software implementations. The computations are programmed into the silicon chip such that an FPGA system can be programmed and reprogrammed many times. The utility of FPGAs does, however, come at a price. Whilst, compared to a microprocessor, FPGAs are typically several orders of magnitude faster and more power efficient, the task of creating efficient programs for these devices is difficult.

Typically, FPGAs are useful only for operations that process large streams of data, such as signal processing, networking, and the like. Compared to integrated circuit, they may be 5 to 25 times worse in terms of area, delay, and performance [18]. However, while an integrated circuit design may take months to years to develop and have a multimillion-dollar price tag, an FPGA design might only take days to create and cost tens to hundreds of dollars.

## 3.2 Field-Programmable Gate Array (FPGA)

## 3.3 Application-Specific Integrated Circuit (ASIC)

## 3.4 Hardware Description Languages

### 3.4.1 VHSIC Hardware Description Language (VHDL)

### 3.4.2 Verilog (IEEE 1364)

### 3.4.3 High Level Synthesis

High level synthesis is a process which is able to transform a design specification from a low-level programming language (such as C, C++ or SystemC) into an Register Transfer Level (RTL) implementation. In particular, this thesis will focus on the “Xilinx AutoESL” High Level Synthesis tool.

The design flow of AutoESL comprises three major stages [5]:

**Synthesis** Creates an RTL implemenetation from the source code.

**Simulation** Verifies the RTL through co-simulation with a test bench.

**Implementation** Generates and executes scripts to perform logic analysis.

# Chapter 4

# Design

## 4.1 Introduction

## 4.2 Algorithm Profiling

In order to choose which step/steps of the outlier detection algorithm would be best suited for implementation on FPGA hardware, it was first necessary to profile the execution of the algorithm using various test cases. This task was performed using MATLAB, using code supplied by Khoa that was used to test and verify the conclusions of "Large Scale Anomaly Detection and Clustering Using Random Walks". Using *MATLAB*'s `profile` command, I was able to analyse the algorithm and make an assessment of the running time of the algorithm.

The results of the algorithm profiling appear in the following sections.

### 4.2.1 Performance Bottleneck

From observations of the results of the algorithm profiling, it was observed that the performance of the 'anomaly detection using commute-distance' algorithm is bottle-necked significantly by a function named `TopN_Outlier_Pruning_Block`. The MATLAB code for this function can be found in Appendix B. Analysis of this function, as well as discussions with Khoa revealed that the algorithm was

| **Name** | **Size ($M$)** | **Dimensionality ($N$)** | **Comments** |
|---|---|---|---|
| testCD | 640 | 2 | |
| testCDST | 2000 | 2 | |
| testCDST2 | 2000 | 2 | |
| testCDST3 | 2000 | 2 | |
| testoutrank | 441 | 2 | |
| pendigits | 4601 | 58 | |
| musk | 67557 | 43 | |
| connect4 | 10992 | 17 | |
| spam | 6598 | 167 | |

**Table 4.1:** Data set descriptions

**Algorithm 1:** TopN_Outlier_Pruning_Block

```
k: the number of nearest neighbors
N: the number of outliers to return
Data: a set of examples in random order
outliers: a set of outliers
begin
    cutoff ⟵ 0; // set the cutoff for pruning to 0
    outliers ⟵ ∅; // initialize to the empty set
    while block ⟵ getNextBlock(Data) do // load a block of
    examples from d
        neighbours(b) ⟵ ∅,   ∀ b ∈ block;
        foreach d ∈ Data do
            foreach b ∈ block : b ≠ d do
                if |neighbours(b)| < k ∨ distance(b, d) <
                maxDist(b, neighbours (b)) then
                    neighbours(b) ⟵ closest(b, neighbours (b)) ∪ d, k);
                    if score(neighbours(b), b) < cutoff then
                        block ⟵ block \ b;
                    end
                end
            end
        end
        outliers ⟵top(block ∪ outliers, N); // keep only the top n
        outliers
        cutoff ⟵ min_{o∈outliers}(score(o)); // the cutoff is the score
        of the weakest outlier
    end
    return outliers;
end
```

originally devised by Bay and Schwabacher and published in the paper "Mining distance-based outliers in near linear time with randomization and a simple pruning rule". The general steps of the algorithm are described in algorithm 1.

In this algorithm, the `score` function can be any monotonically decreasing function of the nearest neighbor distances such as the distance to the $k$th nearest neighbor, or the average distance to the $k$ neighbours [7].

The main idea of the nested loop algorithm is that for each example in the input set `Data`, the algorithm keeps track of the closest neighbours found so far. When an example's closest neighbours achieve a score lower than the `cutoff`, the example is discarded because it can no longer be an outlier. As more examples are processed, the algorithm finds more extreme outliers and the `cutoff` increases along with pruning efficiency [7].

In the worst case, the performance of the algorithm is very poor — due to the nested loops, it could require $O(N^2)$ distance computations and $O(\frac{N}{blocksize} \times N)$ data accesses. However, Bay and Schwabacher proved (through application of the algorithm to both real and synthetic data sets) that the algorithm performs considerably better than the expected $O(N^2)$ running time in the average case. The performance improvements over similar algorithms were attributed to the application of randomization and pruning techniques. The outlier pruning problem can be considered similar to the problem of conducting a set of independent Bernoulli trials in which examples are analysed until $k$ examples within distance $d$ are found, or until the data set is exhausted. Bay and Schwabacher proved that the number of trials expected to achieve $k$ examples within distance $d$ is given by:

$$E[Y] \leq \frac{k}{\pi(\mathbf{x})} + \left(1 - \sum_{y=k}^{N} P(Y = y)\right) \times N \tag{4.1}$$

Where $\pi(x)$ is the probability that a random drawn example lies within distance $d$ of point $x$ and $P(Y = y)$ is the probability of obtaining the $k$th success on trial $y$.

The first term in Equation 4.1 represents the number of distance computations to eliminate non-outliers, and does not depend on $N$. The second term represents the expected cost of outliers, and does depend on $N$, yielding an overall quadratic dependency to process $N$ examples in total. However, note that we typically set the program parameters to return a small and possibly fixed number of outliers. Thus the first term dominates and we obtain near linear performance [7]. More specifically, it was determined that the primary factor determining the scaling is how the cutoff changes as $N$ increases.

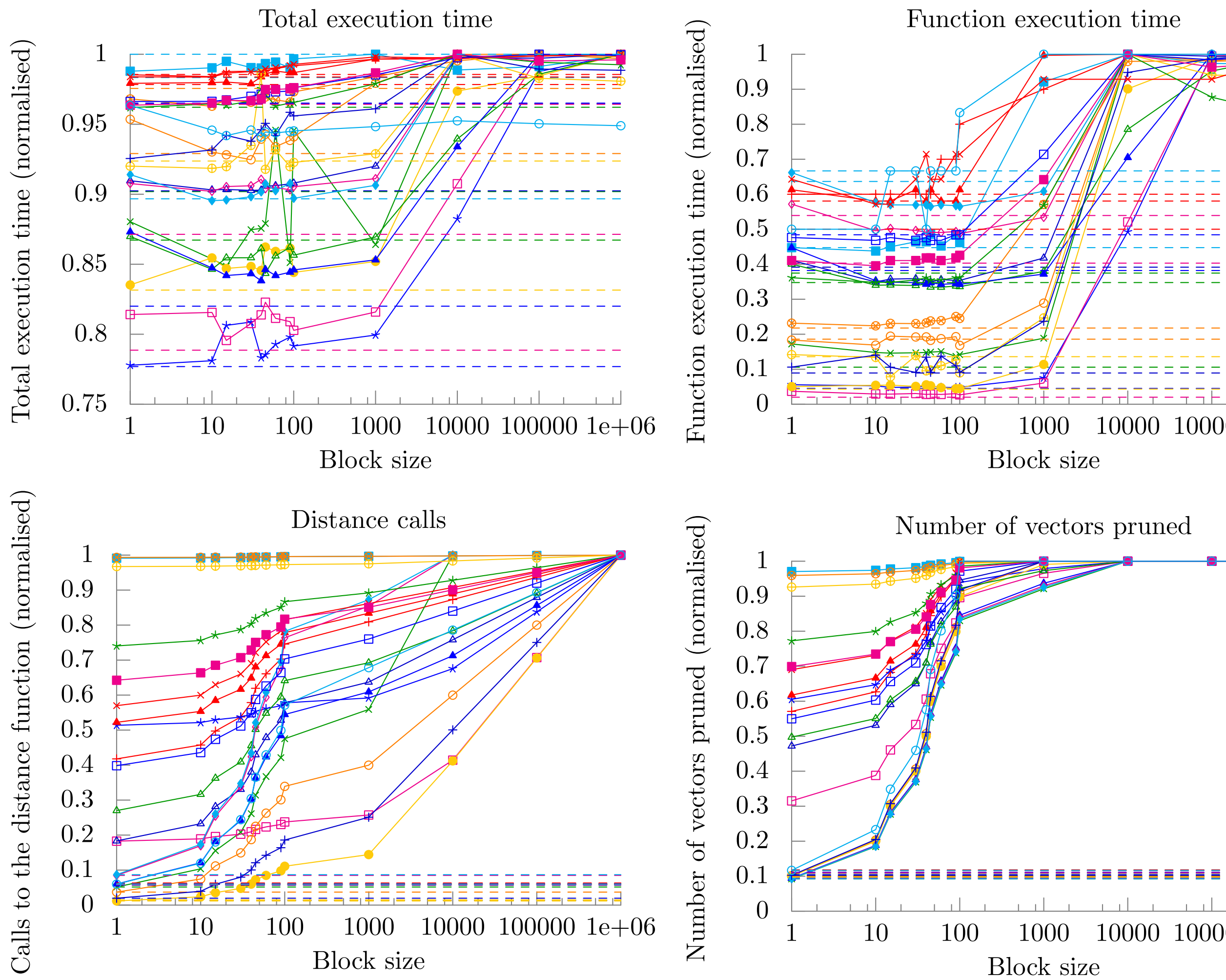
There are, however, some limitations of the aforementioned algorithm. Specifically:

1. The algorithm assumes that the data is in random order. If the data is not in random order and is sorted then the performance can be poor.

2. The algorithm depends on the independence of examples. If examples are dependent in such a way that they have similar values (and will likely be in the set of $k$ nearest neighbors) this can cause performance to be poor as the algorithm may have to scan the entire data set to find the dependent examples.

3. The algorithm can perform poorly occurs when the data does not contain outliers.

## 4.3 The effects of blocking and a comparison of block sizes



# Chapter 5

# Implementation

# Chapter 6

# Results

# Chapter 7

# Conclusions

# Appendix A

# Computer specifications

All test results contained in this thesis were performed on a server with the following specifications. The following specifications were obtained using the "Hardware Lister" (`lshw`) tool.

| Item | Specification |
|---|---|
| **CPU** | |
| **Product** | Intel(R) Core(TM) i5 CPU M 540 @ 2.53GHz |
| **Capacity** | 1199MHz |
| **Width** | 64 bits |
| **Clock** | 133MHz |
| **Cores** | 2 |
| **L1 cache** | 32KiB (asynchronous internal write-through data) |
| **L2 cache** | 256KiB (burst internal write-through unified) |
| **L3 cache** | 8MiB (burst internal write-back) |
| **Memory** | |
| **Description** | SODIMM DDR3 Synchronous 1066 MHz |
| **Size** | 4GiB |
| **Width** | 64 bits |
| **Clock** | 1066MHz |

**Table A.1:** Hardware specifications

| Item | Specification |
|---|---|
| **gcc** | gcc (Ubuntu/Linaro 4.6.1-9ubuntu3) 4.6.1 |
| **ld** | GNU ld (GNU Binutils for Ubuntu) 2.21.53.20110810 |
| **MATLAB** | |
| **mex** | |

**Table A.2:** Software specifications

| Item | Specification |
|---|---|
| **Operating System** | Ubuntu 11.10 |
| **Platform** | x86_64 |
| **Kernel version** | 3.0.0-17-generic |

**Table A.3:** Operating System specifications

# Appendix B

# MATLAB Code

```
% Find the top N outliers by comparing average distances
% to the k nearest neighbours with pruning technique.
function [O,OF]=TopN_Outlier_Pruning_Block(X,k,N,
    block_size)
  n = size(X,1);
  OF = zeros(1,N);
  O = OF;

  c = 0;
  count = 0;
  while (n-count > 0)
    B = zeros(1,block_size);
    B = (count+1 : count+block_size);
    count = count+block_size;

    if count <= n
      sizeB = block_size;
    else
      sizeB = n-(count-block_size);
    end

    neighbours = zeros(sizeB,k);
    neighbours_dist = zeros(sizeB,k);
    score = zeros(1,sizeB);

    l = 1;
    for i = 1:n
      for j = 1:sizeB
        if i~=B(j) && B(j)~=0
          d=euclidean_dist_squared(X(i,:),X(B(j),:));

          if l>1 && l<=k+1 && neighbours(j,l-1)==0
            l=l-1;
          elseif l<k && neighbours(j,l)~=0
```

```
            l=l+1;
          end

          if l<=k
            neighbours(j,l)=i;
            neighbours_dist(j,l)=d;
            if l==k
              score(j)=mean(neighbours_dist(j,:),2);
            end
          else % l>k
            % find the farthest point
            [maxd,maxi]=max(neighbours_dist(j,:));

            % replace the farthest point
            if d<maxd
              neighbours(j,maxi)=i;
              neighbours_dist(j,maxi)=d;

              % update the score
              score(j)=(score(j)*k-maxd+d)/k;
              if score(j)<=0
                % avoid round off error
                score(j)=max(mean(neighbours_dist(j,:),2)
                    ,0);
              end
            end
          end

          if l>=k && score(j)<c
            B(j)=0;
            %neighbours(j,:)=0;
            %neighbours_dist(j,:)=0;
            score(j)=0;
          end
        end
      end
      l=l+1;
    end

    % O=Top(B U O,N)
    BO=[B(1:sizeB) O];
    BOF=[score OF];
    [BOF,index]=sort(BOF,'descend');
    BO=BO(index);

    O=BO(1:N);
    OF=BOF(1:N);

    %c=weakest outlier
    c=OF(N);
```

```
end
```

# Appendix C

# C Code

```
/* Includes */
#include <float.h> /* for DBL_MAX */
#include <string.h> /* for memset, memcpy */
#include "macros.h"
#include "top_n_outlier_pruning_block.h"

/* Check compatibility of defined macros. */
#if (defined(UNSORTED_INSERT) && defined(SORTED_INSERT))
    || (!defined(UNSORTED_INSERT) && !defined(
    SORTED_INSERT))
  #error "Exactly one of UNSORTED_INSERT and
      SORTED_INSERT should be defined."
#endif /* #if defined(UNSORTED_INSERT) && defined(
    SORTED_INSERT) */

#if !defined(DEBUG) && defined(STATS)
  #error "STATS should only be defined in DEBUG mode."
#endif /* #if !defined(DEBUG) && defined(STATS) */

/* Forward declarations */
static inline double_t distance_squared(
  const double_t * const vector1,
  const double_t * const vector2,
  const size_t vector_dims
  );
static inline double_t insert(
  index_t * const outliers,
  double_t * const outlier_scores,
  const size_t k,
  uint_t * const found,
  const index_t new_outlier,
  const double_t new_outlier_index
  );
static inline void best_outliers(
```

```
  index_t * const outliers,
  double_t * const outlier_scores,
  size_t * outliers_size,
  const size_t N,
  index_t * const current_block,
  double_t * const scores,
  const size_t block_size
  );
static inline void sort_vectors_descending(
  index_t *  const current_block,
  double_t * const scores,
  const size_t block_size
  );
static inline void merge(
  index_t * const global_outliers,
  double_t * const global_outlier_scores,
  const size_t global_outliers_size,
  const size_t N,
  index_t * const local_outliers,
  double_t * const local_outlier_scores,
  const size_t block_size,
  index_t * const new_outliers,
  double_t * const new_outlier_scores,
  size_t * new_outliers_size
  );

#ifdef STATS
static lint_t calls_counter = 0;
static uint_t num_pruned = 0;

void get_stats(lint_t * const counter, uint_t * const prune_count) {
  ASSERT_NOT_NULL(counter);
  ASSERT_NOT_NULL(prune_count);

  *counter = calls_counter;
  *prune_count = num_pruned;
}
#endif /* #ifdef STATS */

static inline double_t distance_squared(const double_t * const vector1, const double_t * const vector2, const size_t vector_dims) {
  ASSERT_NOT_NULL(vector1);
  ASSERT_NOT_NULL(vector2);
  ASSERT(vector_dims > 0);

#ifdef STATS
  const UNUSED lint_t old_calls_counter = calls_counter;
  calls_counter++;
```

```c
  ASSERT(calls_counter > old_calls_counter);
#endif /* #ifdef STATS */

  double_t sum_of_squares = 0;

  uint_t dim;
  for (dim = 0; dim < vector_dims; dim++) {
    const double_t val = vector1[dim] - vector2[dim];
    const double_t val_squared = val * val;
    sum_of_squares += val_squared;
  }

  return sum_of_squares;
}

static inline double_t insert(index_t * const outliers,
                              double_t * const outlier_scores,
                              const size_t k,
                              uint_t * const found,
                              const index_t new_outlier,
                              const double_t new_outlier_score) {
  /* Error checking. */
  ASSERT_NOT_NULL(outliers);
  ASSERT_NOT_NULL(outlier_scores);
  ASSERT(k > 0);
  ASSERT_NOT_NULL(found);
  ASSERT(*found <= k);
  ASSERT(new_outlier >= start_index);

  int_t insert_index  = -1; /* the index at which the new outlier will be inserted */
  double_t removed_value = -1; /* the value that was removed from the outlier_scores array */

#if defined(SORTED_INSERT)
  /*
   * Shuffle array elements from front to back. Elements greater than the new
   * value will be right-shifted by one index in the array.
   *
   * Note that uninitialised values in the array will appear on the left. That
   * is, if the array is incomplete (has a size n < N) then the data in the
   * array is stored in the rightmost n indexes.
   */

  if (*found < k) {
```

```
    /* Special handling required if the array is incomplete. */

    uint_t i;
    for (i = k - *found - 1; i < k; i++) {
      if (new_outlier_score > outlier_scores[i] || i == (k - *found - 1)) {
        /* Shuffle values down the array. */
        if (i != 0) {
          outliers[i-1] = outliers[i];
          outlier_scores[i-1] = outlier_scores[i];
        }
        insert_index  = i;
        removed_value = 0;
      } else {
        /* We have found the insertion point. */
        break;
      }
    }
  } else {
    int_t i;
    for (i = k-1; i >= 0; i--) {
      if (new_outlier_score < outlier_scores[i]) {
        if ((unsigned) i == k-1)
          /*
           * The removed value is the value of the last element in the
           * array.
           */
          removed_value = outlier_scores[i];

        /* Shuffle values down the array. */
        if (i != 0) {
          outliers[i] = outliers[i-1];
          outlier_scores[i] = outlier_scores[i-1];
        }
        insert_index = i;
      } else {
        /* We have found the insertion point. */
        break;
      }
    }
  }
#elif defined(UNSORTED_INDEX)
  if (*found < k) {
    insert_index = *found + 1;
    removed_value = 0;
  } else {
    int_t max_index = -1;
    double_t max_value = DBL_MAX;
```

```

    int_t i;
    for (i = k-1; i >= 0; i--) {
      if (max_index <= 0 || outlier_scores[i] > max_value
          ) {
        max_index = i;
        max_value = outlier_scores[i];
      }
    }

    if (new_outlier_score < max_value) {
      insert_index  = max_index;
      removed_value = max_value;
    }
  }
#endif /* #if defined(SORTED_INSERT) */

  /*
   * Insert the new pair and increment the current_size
      of the array (if
   * necessary).
   */
  if (insert_index >= 0) {
    outliers[insert_index] = new_outlier;
    outlier_scores[insert_index] = new_outlier_score;

    if (*found < k)
      (*found)++;
  }

  return removed_value;
}

static inline void best_outliers(index_t * const outliers
    ,
                  double_t * const outlier_scores,
                  size_t * outliers_size,
                  const size_t N,
                  index_t * const current_block,
                  double_t * const scores,
                  const size_t block_size) {
  /* Error checking. */
  ASSERT_NOT_NULL(outliers);
  ASSERT_NOT_NULL(outlier_scores);
  ASSERT_NOT_NULL(outliers_size);
  ASSERT(*outliers_size <= N);
  ASSERT(N > 0);
  ASSERT_NOT_NULL(current_block);
  ASSERT_NOT_NULL(scores);
  ASSERT(block_size > 0);
```

```c

  /* Sort the (current_block, scores) vectors in
     descending order of value. */
  sort_vectors_descending(current_block, scores,
     block_size);

  /* Create two temporary vectors for the output of the "
     merge" function. */
  index_t  new_outliers[N];
  double_t new_outlier_scores[N];
  size_t   new_outliers_size = 0;

  memset(new_outliers, null_index, N * sizeof(index_t));
  memset(new_outlier_scores, 0, N * sizeof(double_t));

  /* Merge the two vectors. */
  merge(outliers, outlier_scores, *outliers_size, N,
     current_block, scores, block_size, new_outliers,
     new_outlier_scores, &new_outliers_size);

  /* Copy values from temporary vectors to real vectors.
     */
  memcpy(outliers, new_outliers, N * sizeof(index_t));
  memcpy(outlier_scores, new_outlier_scores, N * sizeof(
     double_t));
  *outliers_size = new_outliers_size;
}

static inline void sort_vectors_descending(index_t *
    const current_block,
                         double_t * const scores,
                         const size_t block_size) {
  /* Error checking. */
  ASSERT_NOT_NULL(current_block);
  ASSERT_NOT_NULL(scores);
  ASSERT(block_size > 0);

  uint_t i;
  for (i = 0; i < block_size; i++) {
    int_t j;
    index_t  ind = current_block[i];
    double_t val = scores       [i];
    for (j = i-1; j >= 0; j--) {
      if (scores[j] >= val)
        break;
      current_block[j+1] = current_block[j];
      scores       [j+1] = scores       [j];
    }
    current_block[j+1] = ind;
    scores       [j+1] = val;
```

```c
  }
}

static inline void merge(index_t * const global_outliers, double_t * const global_outlier_scores, const size_t global_outliers_size, const size_t N,
              index_t * const local_outliers,  double_t * const local_outlier_scores,  const size_t block_size,
              index_t * const new_outliers, double_t * const new_outlier_scores, size_t * new_outliers_size) {
  /* Error checking. */
  ASSERT_NOT_NULL(global_outliers);
  ASSERT_NOT_NULL(global_outlier_scores);
  ASSERT(global_outliers_size <= N);
  ASSERT(N > 0);
  ASSERT_NOT_NULL(local_outliers);
  ASSERT_NOT_NULL(local_outlier_scores);
  ASSERT(block_size > 0);
  ASSERT_NOT_NULL(new_outliers);
  ASSERT_NOT_NULL(new_outlier_scores);
  ASSERT_NOT_NULL(new_outliers_size);

  *new_outliers_size  = 0;
  uint_t iter = 0; /* iterator through output array */
  uint_t global = 0; /* iterator through global array */
  uint_t local = 0; /* iterator through local array */
  while (iter < N && (global < global_outliers_size || local < block_size)) {
    if (global >= global_outliers_size && local < block_size) {
      /* There are no remaining elements in the global arrays. */
      new_outliers[iter] = local_outliers[local];
      new_outlier_scores[iter] = local_outlier_scores[local];
      local ++;
      global++;
    } else if (global < global_outliers_size && local >= block_size) {
      /* There are no remaining elements in the local arrays. */
      new_outliers[iter] = global_outliers[global];
      new_outlier_scores[iter] = global_outlier_scores[global];
      local ++;
      global++;
    } else if (global >= global_outliers_size && local >= block_size) {
```

```
      /*
       * There are no remaining elements in either the global or local
       * arrays.
       */
      break;
    } else if (global_outlier_scores[global] >= local_outlier_scores[local]) {
      new_outliers[iter] = global_outliers[global];
      new_outlier_scores[iter] = global_outlier_scores[global];
      global++;
    } else if (global_outlier_scores[global] <= local_outlier_scores[local]) {
      new_outliers[iter] = local_outliers[local];
      new_outlier_scores[iter] = local_outlier_scores[local];
      local++;
    }

    iter++;
    (*new_outliers_size)++;
  }
}

void top_n_outlier_pruning_block(const double_t * const data,
                   const size_t num_vectors, const size_t vector_dims,
                   const size_t k, const size_t N, const UNUSED size_t default_block_size,
                   index_t * outliers, double_t * outlier_scores) {
  /* Error checking. */
  ASSERT_NOT_NULL(data);
  ASSERT(vector_dims > 0);
  ASSERT(k > 0);
  ASSERT(N > 0);
  ASSERT(default_block_size > 0);
  ASSERT_NOT_NULL(outliers);
  ASSERT_NOT_NULL(outlier_scores);

  /* Set output to zero. */
  memset(outliers, null_index, N * sizeof(index_t));
  memset(outlier_scores, 0, N * sizeof(double_t));

  double_t cutoff = 0; /* vectors with a score less than the cutoff will be removed from the block */
  size_t   outliers_found = 0; /* the number of initialised elements in the outliers array */
```

```c

#ifndef NO_BLOCKING
  index_t  block_begin; /* the index of the first vector in the block currently being processed */
  size_t   block_size; /* block_size may be smaller than devfault_block_size if "num_vectors mod default_block_size != 0" */

  for (block_begin = 0; block_begin < num_vectors; block_begin += block_size) { /* while there are still blocks to process */
    block_size = MIN(block_begin + default_block_size, num_vectors) - block_begin; /* the number of vectors in the current block */
    ASSERT(block_size <= default_block_size);

    index_t current_block[block_size]; /* the indexes of the vectors in the current block */
    index_t neighbours[block_size][k]; /* the "k" nearest neighbours for each vector in the current block */
    double neighbours_dist[block_size][k]; /* the distance of the "k" nearest neighbours for each vector in the current block */
    double score[block_size]; /* the average distance to the "k" neighbours */
    uint_t found[block_size]; /* how many nearest neighbours we have found, for each vector in the block */

    /* Reset array contents */
    uint_t i;
    for (i = 0; i < block_size; i++) {
      if (i < block_size)
        current_block[i] = (index_t)((block_begin + i) + start_index);
      else
        current_block[i] = null_index;
    }
    memset(&neighbours, null_index, block_size * k * sizeof(index_t));
    memset(&neighbours_dist, 0, block_size * k * sizeof(double));
    memset(&score, 0, block_size * sizeof(double));
    memset(&found, 0, block_size * sizeof(uint_t));

    index_t vector1;
    for (vector1 = start_index; vector1 < num_vectors + start_index; vector1++) {
      uint_t block_index;
```

```
    for (block_index = 0; block_index < block_size; block_index++) {
      const index_t vector2 = current_block[block_index];

      if (vector1 != vector2 && vector2 >= start_index) {
        /*
         * Calculate the square of the distance between the two
         * vectors (indexed by "vector1" and "vector2")
         */
        const double_t dist_squared = distance_squared(&data[(vector1-start_index) * vector_dims], &data[(vector2-start_index) * vector_dims], vector_dims);

        /*
         * Insert the new (index, distance) pair into the neighbours
         * array for the current vector.
         */

        const double_t removed_distance = insert(neighbours[block_index], neighbours_dist[block_index], k, &found[block_index], vector1, dist_squared);

        /*
         * Update the score (if the neighbours array was changed).
         */
        if (removed_distance >= 0)
          score[block_index] = (double_t) ((score[block_index] * k - removed_distance + dist_squared) / k);

        /*
         * If the score for this vector is less than the cutoff,
         * then prune this vector from the block.
         */
        if (found[block_index] >= k && score[block_index] < cutoff) {
          current_block[block_index] = null_index;
          score[block_index] = 0;
#ifdef STATS
          const UNUSED uint_t old_num_pruned = num_pruned;
          num_pruned++;
```

```c
            ASSERT(num_pruned > old_num_pruned);
#endif /* #ifdef STATS */
          }
        }
      }
    }

    /* Keep track of the top "N" outliers. */
    best_outliers(outliers, outlier_scores, &outliers_found, N, current_block, score, block_size);

    /*
     * Set "cutoff" to the score of the weakest outlier. There is no need to
     * store an outlier in future iterations if its score is better than the
     * cutoff.
     */
    cutoff = outlier_scores[N-1];
  }
#else
  index_t vector1;
  for (vector1 = start_index; vector1 < num_vectors + start_index; vector1++) {
    index_t neighbours[k]; /* the "k" nearest neighbours for the current vector */
    double_t neighbours_dist[k]; /* the distance of the "k" nearest neighbours for the current vector */
    double_t score = 0; /* the average distance to the "k" neighbours */
    uint_t found = 0; /* how many nearest neighbours we have found */
    boolean removed = false; /* true if vector1 has been pruned */

    memset(neighbours, null_index, k * sizeof(index_t));
    memset(neighbours_dist, 0, k * sizeof(double_t));

    index_t vector2;
    for (vector2 = start_index; vector2 < num_vectors + start_index && !removed; vector2++) {
      if (vector1 != vector2) {
        /*
         * Calculate the square of the distance between the two
         * vectors (indexed by "vector1" and "vector2")
         */
        const double_t dist_squared = distance_squared(&data[(vector1-start_index) * vector_dims], &
```

```
              data[(vector2-start_index) * vector_dims],
              vector_dims);

        /*
         * Insert the new (index, distance) pair into the
            neighbours
         * array for the current vector.
         */
        const double_t removed_distance = insert(
              neighbours, neighbours_dist, k, &found,
              vector2, dist_squared);

        /* Update the score (if the neighbours array was
              changed). */
        if (removed_distance >= 0)
          score = (double_t) ((score * k -
              removed_distance + dist_squared) / k);

        /*
         * If the score for this vector is less than the
            cutoff,
         * then prune this vector from the block.
         */
        if (found >= k && score < cutoff) {
          removed = true;
#ifdef STATS
          const UNUSED uint_t old_num_pruned = num_pruned
              ;
          num_pruned++;
          ASSERT(num_pruned > old_num_pruned);
#endif /* #ifdef STATS */
          break;
        }
      }
    }

    if (!removed) {
      /* Keep track of the top "N" outliers. */
      best_outliers(outliers, outlier_scores, &
          outliers_found, N, &vector1, &score, 1);

      /*
       * Set "cutoff" to the score of the weakest outlier
          . There is no need to
       * store an outlier in future iterations if its
          score is better than the
       * cutoff.
       */
      cutoff = outlier_scores[N-1];
    }
```

```
    }
#endif /* #ifndef NO_BLOCKING */
}
```

# Appendix D

# AutoESL code

# Bibliography


[1] *AutoESL Coding Style Guide*. Apr. 2012. 160 pp.

[2] *AutoESL Operator and Core Guide*. Apr. 2012. 18 pp.

[3] *AutoESL Reference Guide*. Apr. 2012. 125 pp.

[4] *AutoESL Tutorial: Integrating with EDK*. Apr. 2012. 125 pp.

[5] *AutoESL Tutorial: Introduction*. Apr. 2012. 71 pp.

[6] *AutoESL User Guide*. Apr. 2012. 183 pp.

[7] Stephen D. Bay and Mark Schwabacher. "Mining distance-based outliers in near linear time with randomization and a simple pruning rule". In: *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. KDD 2003. Washington, D.C.: ACM, Aug. 2003, pp. 29–38. ISBN: 1-58113-737-0. DOI: 10.1145/956750.956758. URL: http://doi.acm.org/10.1145/956750.956758 (visited on 21/05/2012).

[8] Berkeley University of California. *Graph Partitioning, Part 2*. CS267 Lecture Notes. Apr. 1999. URL: http://www.cs.berkeley.edu/~demmel/cs267/lecture20/lecture20.html (visited on 01/05/2012).

[9] Varun Chandola, Arindam Banerjee and Vipin Kumar. "Anomaly Detection: A Survey". In: *ACM Computing Serveys* 41.3 (July 2009), 15:1–15:58. ISSN: 0360-0300. URL: http://www.cs.umn.edu/tech_reports_upload/tr2007/07-017.pdf (visited on 01/05/2012).

[10] Ashok K. Chandra et al. "The Electrical Resistance of a Graph Captures its Commute and Cover Times". In: *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*. STOC 1989. New York, NY, USA: ACM, 1989, pp. 574–586. ISBN: 0-89791-307-8. DOI: 10.1145/73007.73062. URL: http://doi.acm.org/10.1145/73007.73062 (visited on 03/05/2012).

[11] Peter G. Doyle and J. Laurie Snell. *Random Walks and Electric Networks*. Washington, DC: Mathematical Association of America, 2006.

[12] Ganglong Duan, Zhiwen Huang and Jianren Wang. "Extreme Learning Machine for Financial Distress Prediction for Listed Company". In: *2010 International Conference on Logistics Systems and Intelligent Management*. Vol. 3. IEEE Computer Society, Jan. 2010, pp. 1961–1965. ISBN: 978-1-4244-7331-1. DOI: 10.1109/ICLSIM.2010.5461268. (Visited on 01/05/2012).

[13] Francois Fouss et al. “Random-Walk Computation of Similarities between Nodes of a Graph with Application to Collaborative Recommendation”. In: *IEEE Trans. on Knowl. and Data Eng.* 19.3 (Mar. 2007), pp. 355–369. ISSN: 1041-4347. DOI: `10.1109/TKDE.2007.46`.

[14] Pascal Getreuer. *Writing MATLAB C/MEX Code*. Apr. 2010. 29 pp. URL: `http://www.mathworks.com/matlabcentral/fileexchange/authors/14582` (visited on 13/05/2012).

[15] Ary L. Goldberger et al. “PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals”. In: *Circulation* 101.23 (2000), e215–e220. DOI: `10.1161/01.CIR.101.23.e215`. URL: `http://circ.ahajournals.org/content/101/23/e215.full` (visited on 01/05/2012).

[16] Gene Howard Golub and Charles F. Van Loan. *Matrix Computations*. 3rd ed. Johns Hopkins University Press, 1996. ISBN: 9780801854149.

[17] Charles M. Grinstead and J. Laurie Snell. *Introduction to Probability*. 2nd ed. American Mathematical Society, 1997, pp. 405–470. ISBN: 978-0821807491. URL: `http://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/amsbook.mac.pdf` (visited on 03/05/2012).

[18] Scott Hauck and André DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation (Systems on Silicon)*. 1st ed. Morgan Kaufmann Publishers, 2007. ISBN: 978-0123705228.

[19] Douglas M. Hawkins. *Identification of outliers*. Monographs on Applied Probability and Statistics. Chapman and Hall, 1980. ISBN: 9780412219009.

[20] Cris L. Luengo Hendriks. *MEX-File Programming for Image Processing Using DIPimage*. Mar. 2011.

[21] I.T. Jolliffe. *Principal Component Analysis*. 2nd ed. Springer, 2002. ISBN: 978-0387954424.

[22] Richard Kettlewell. *Inline Functions in C*. URL: `http://http://www.greenend.org.uk/rjk/tech/inline.html` (visited on 26/08/2012).

[23] Nguyen Lu Dang Khoa. “Large Scale Anomaly Detection and Clustering Using Random Walks”. PhD thesis. University of Sydney, Mar. 2012.

[24] Nguyen Lu Dang Khoa and Sanjay Chawla. “Robust Outlier Detection Using Commute Time and Eigenspace Embedding”. In: *PAKDD*. Vol. 2. 2010, pp. 422–434. DOI: `10.1007/978-3-642-13672-6_41`.

[25] Edwin M. Knorr, Raymond T. Ng and Vladimir Tucakov. “Distance-based outliers: algorithms and applications”. In: *The VLDB Journal — The International Journal on Very Large Data Bases* 8.3–4 (Feb. 2000), pp. 237–253. DOI: `10.1007/s007780050006`.

[26] Karen A. Kopecky. *Calling C and Fortran Programs from MATLAB*. URL: `http://www.karenkopecky.net/Teaching/Cclass/MatlabCallsC.pdf` (visited on 13/05/2012).

[27] Shi Lei et al. "Financial Data Mining based on Support Vector Machines and Ensemble Learning". In: *Proceedings of the 2010 International Conference on Intelligent Computation Technology and Automation*. Vol. 2. ICICTA 2010. Washington, DC, USA: IEEE Computer Society, May 2010, pp. 313–314. ISBN: 978-0-7695-4077-1. DOI: 10.1109/ICICTA.2010.787.

[28] Xiuquan Li and Zhidong Deng. "A Machine Learning Approach to Predict Turning Points for Chaotic Financial Time Series". In: *Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence - Volume 02*. ICTAI 2007. Washington, DC, USA: IEEE Computer Society, 2007, pp. 331–335. ISBN: 0-7695-3015-X. DOI: 10.1109/ICTAI.2007.18. URL: http://dx.doi.org/10.1109/ICTAI.2007.18 (visited on 22/08/2012).

[29] L. Lovász. "Random Walks on Graphs: A Survey". In: *Combinatorics, Paul Erdős is Eighty*. Ed. by D. Miklós, V. T. Sós and T. Szőnyi. Vol. 2. Budapest: János Bolyai Mathematical Society, 1996, pp. 353–398.

[30] Oded Z Maimon and Lior Rokach. *Data Mining and Knowledge Discovery Handbook: A Complete Guide for Practitioners and Researchers*. Kluwer Academic Publishers, 2005, pp. 1–16. ISBN: 9780387244358.

[31] MathWorks. *Examples of C/C++ Source MEX-Files*. Product Documentation. URL: http://http://www.mathworks.com.au/help/techdoc/matlab_external/f12977.html (visited on 13/05/2012).

[32] MathWorks. *MEX-files Guide*. URL: http://www.mathworks.com.au/support/tech-notes/1600/1605.html (visited on 13/05/2012).

[33] H. D. K. Moonesinghe and Pang-Ning Tan. "Outrank: a Graph-Based Outlier Detection Framework Using Random Walk". In: *International Journal on Artificial Intelligence Tools* 17.1 (2008), pp. 19–36. DOI: 10.1142/S0218213008003753.

[34] *Active forgetting in machine learning and its application to financial problems*. Vol. 5. 2000, 123–128 vol.5. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=861445 (visited on 22/08/2012).

[35] Sukanta Pati. *Laplacian matrix of a graph*. http://www.lix.polytechnique.fr/~schwander/resources/mig/slides/pati.pdf. Talk prepared for the Indo-French workshop, 2011. 2011. (Visited on 01/05/2012).

[36] Robin Pottathuparambil et al. "Low-latency FPGA Based Financial Data Feed Handler". In: *IEEE International Symposium on Field-Programmable Custom Computing Machines. Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. FCCM 2011. Washington, DC, USA: IEEE Computer Society, May 2011, pp. 93–96. ISBN: 978-0-7695-4301-7. DOI: 10.1109/FCCM.2011.50.

[37] Daniel A. Spielman. "Algorithms, Graph Theory, and Linear Equations in Laplacian Matrices". In: *Proceedings of the International Congress of Mathematicians 2010*. 2010, pp. 2698–2722. DOI: 10.1142/9789814324359.0164. URL: http://www.cs.yale.edu/homes/spielman/PAPERS/icm10post.pdf (visited on 23/05/2012).

[38] Daniel A. Spielman. *The Laplacian*. http://www.cs.yale.edu/homes/spielman/561/lect02-09.pdf. Lecture notes. Sept. 2009. (Visited on 01/05/2012).

[39] Daniel A. Spielman and Shang-Hua Teng. "Nearly-Linear Time Algorithms for Preconditioning and Solving Symmetric, Diagonally Dominant Linear Systems". In: *CoRR* abs/cs/0607105 (Sept. 2006). URL: `http://arxiv.org/pdf/cs/0607105v4.pdf` (visited on 01/05/2012).

[40] Pei Sun and Sanjay Chawla. "On Local Spatial Outliers". In: *ICDM*. 2004, pp. 209–216. DOI: `10.1109/ICDM.2004.10097`.

[41] Andreas Uhl. *Calling C from Matlab: Introduction*. Presentation. URL: `http://www.cosy.sbg.ac.at/~uhl/C-matlab.pdf` (visited on 13/05/2012).

[42] Timothy de Vries, Sanjay Chawla and Michael Houle. "Finding Local Anomalies in Very High Dimensional Space". In: *10th IEEE International Conference on Data Mining*. 2010, pp. 128–137. DOI: `10.1109/ICDM.2010.151`.

[43] Timothy de Vries, Sanjay Chawla and Michael E. Houle. "Density-preserving projections for large-scale local anomaly detection". In: *Knowledge and Information Systems* (June 2011). DOI: `10.1007/s10115-011-0430-4`.

[44] Jean E. Vuillemin et al. In: *IEEE Transactions on VLSI Systems. Readings in hardware/software co-design*. Ed. by Giovanni De Micheli, Rolf Ernst and Wayne Wolf. Norwell, MA, USA: Kluwer Academic Publishers, 2002. Chap. Programmable active memories: reconfigurable systems come of age, pp. 611–624. ISBN: 1-55860-702-1.

[45] Paul D. Yoo, Maria H. Kim and Tony Jan. "Financial Forecasting: Advanced Machine Learning Techniques in Stock Market Analysis". In: *9th International Multitopic Conference*. IEEE. Dec. 2005, pp. 1–7. ISBN: 0-7803-9429-1. DOI: `10.1109/INMIC.2005.334420`.

[46] Lean Yu et al. "Evolving Least Squares Support Vector Machines for Stock Market Trend Mining". In: *Trans. Evol. Comp* 13.1 (Feb. 2009), pp. 87–102. ISSN: 1089-778X. DOI: `10.1109/TEVC.2008.928176`.