DOI: http://dx.doi.org/10.7551/978-0-262-31709-2-ch133

Accelerating Immunos 99

Paul Taylor[1,2] and Fiona A. C. Polack[2] and Jon Timmis[2,3]

[1] BT Research & Innovation, Adastral Park, Martlesham Heath, UK, IP5 3RE
[2] Department of Computer Science, University of York, UK, YO10 5DD
[3] Department of Electronics, University of York, UK, YO10 5DD
paul.n.taylor@bt.com, {fiona.polack,jon.timmis}@york.ac.uk

Abstract

Immunos 99 is a classification algorithm based upon the principles of Artificial Immune Systems (AIS). AIS algorithms can provide alternatives to classical techniques such as decision trees for classification tasks. Immunos 99 provides one alternative however the algorithm and implementations have some room for performance improvement. This paper discusses improvements made to Immunos 99 and its reference implementation to improve runtime performance. The new algorithm/implementation results in an approximate 40% reduction in run time on test data. The paper closes with a proposal for an implementation of the Immunos 99 algorithm that is suitable for use on map/reduce clusters.

Introduction

The classification of large data sets is increasingly important in a wide variety of applications. Many classification techniques are available, with varying applicability. The need to apply classification techniques to large datasets has motivated research on improving the efficiency or running time of algorithms. For example the AIRS classifier (Watkins, Timmis, & Boggess, 2004) has been successfully parallelised, reducing execution time while retaining classification performance (Watkins & Timmis, 2004).

Our interest in the classification of large data sets relates to the automation of business process analysis. A business process is a set of activities that are required in a specific order, to provide a product or service. Efficient organisation of business processes is essential to the efficient running of the business itself. The data collected by monitoring of business processes is not clean, which limits the range of applicable classification techniques (Taylor, Leida, & Majeed, 2012). Techniques which are both resistant to noise in training data and applicable across a wide variety of input data are therefore of particular interest in this endeavour.

In initial small-scale experimentation on samples of poor quality data from real business processes, the Immunos 99 algorithm, proposed by Brownlee (Brownlee, 2005), showed promising performance. However the runtime performance of this (and other) classifier made its use infeasible for large data sets. Immunos 99 is a classification algorithm inspired by the Immunos family, which incorporates additional immune-inspired techniques such as cell proliferation and hypermutation. A complete specification for Immunos 99 can be found in (Brownlee, 2005, Section 5). The only publicly-available implementation of the algorithm is a plugin for the Weka data mining toolkit (Hall et al., 2009; Brownlee, n.d.).

Motivation & Experimental Environment

We are working on monitoring data for business processes relating to provision of service and customer interactions. The ability to predict and prevent the failure of business processes in these areas would enable organisations to perform more efficiently, achieving improved levels of customer service and customer satisfaction. There is little research on the automated analysis of such business processes, not least because of the poor quality of data available. However, if classification is to be useful industrially, it has to be achieved fast enough to allow timely intervention in failing business processes.

In small-scale trials, we tested a range of classifiers, seeking effective prediction of the outcome of a business process is considered; since the business process outcome can be only `SUCCESS` or `FAILURE`, based on some business condition, the problem is a two-class classification task. We find that Immunos 99 can be trained to handle classification of the real data, which is noisy and of low quality.

We next trained Immunos 99 with a typical dataset from BT process monitoring. The proprietary dataset comprises observations on 65,000 business processes, over 46 variables; 74% of the business processes were identified as belonging to the class `SUCCESS`, and the remaining 26% were classified as `FAILURE`. The trial performed 10-fold cross-validation on the dataset, for each classifier under test. However, the execution of the training runs was infeasibly long, with typical executions taking around 40 hours to complete classification of a day's test data.

Since Immunos 99 was the most successful classification technique in small-scale trials, we decided to improve the implementation of the algorithm, exploiting multi-threading, with the goal of reducing execution time. What constitutes an appropriate execution time in this work is dependent on business context – the execution must complete in time to be able to intervene in a business process that may take hours, or may take weeks to complete. Our initial goal was to reduce execution time so that training on one day's data takes hours rather than days.

The purpose and goal of our work on the Immunos 99 algorithm is to enable training and classification of large, noisy data sets, at an appropriate speed for the business context. We do *not* include rigorous algorithmic analysis, as this is not appropriate for our business context.

The improvements discussed here have been tested across large proprietary datasets, and the improvement in execution time has been shown throughout. Results from one of the proprietary datasets is included, to show improvements in execution time are achievable on the real datasets. However, owing to the proprietary nature of the data, the paper illustrates performance by application to a public dataset (Section), allowing detailed discussion and analysis, as well as presentation of reproducible results.

Analysing the Original Immunos 99

Immunos 99 is an immune-inspired classifier originally described by Brownlee (Brownlee, 2005) (Algorithm 1). It utilises a number of immune system features, and analogues of immune system components in its design. In an immune system, an antigen is something that provokes a response from the immune system; in this case, an antigen is analogous to an observation from the input data. In an immune system, the B-cells are recognisers that bind to specific antigens with a particular affinity; here the analogy is of recognisers with an affinity represented as a numerical value.

Here, we use complexity functions to reason about relative execution times, independent of implementation variations. Assuming that each operation in the function takes 1 unit of time, the estimated runtime for a sequential execution of the algorithm is equal to its complexity. Changes to the defined functions to account for the expected reduction in runtime caused by the application of multi-threading and parallelism are discussed in the following section. The complexity functions are derived from examination of the original implementation of the algorithm in (Brownlee, n.d.).

To calculate the complexity function, let g represent the number of generations to use and n represent the number of observations in the input data, which is also the size of the initial B-cell pool. The algorithm performs $g * n^2$ affinity calculations. After each B-cell has been exposed and the affinity calculated, fitness is determined by the B-cell's rank position in the B-cell pool. To determine the rank position requires a sort operation, of complexity in the order of $O(n\ log(n))$.

To train the algorithm, n is the size of the training set. There are c classification classes, and a total of a antibodies in pool, where a_c is the number of antibodies for class c. The approximate complexity of the training phase of Immunos 99 is formed of the three parts: the initial generation of antibodies in equation 1 (line 3); the generation and mutation phase in equation 2 (lines 5-9); and the final pruning phase in 3 (line 12). The complexity of the overall training implementation is the sum of the three phases – the function shows a typical quadratic relationship, plotted against size.

Algorithm 1: Original Immunos 99 Training Algorithm, from (Brownlee, 2005)

```
1  Divide data into antigen groups (by classification label)
2  foreach group do
3      Create initial B-Cell population
4      foreach generation (to parameter limit) do
5          Expose all B-cells to all antigens from all
           groups
6          Calculate fitness scoring
7          Perform population pruning
8          Perform affinity maturation
9          Insert randomly selected antigens of the same
           group
10     end
11 end
12 Perform final pruning for each B-cell population
13 return the final B-cell population as the classifier
```

$$2n + a + c \tag{1}$$

$$g(c(a_c + n(7a_c + 2(a_c log(a_c))))) \tag{2}$$

$$c(a_c + n(3a_c + a_c log(a_c))) \tag{3}$$

Parallelising the Algorithm

A number of changes have been made, to improve training times for the Immunos 99 algorithm. These improvements fall into 2 categories, first partial threading of the algorithm itself, and second changes to the Java implementation (published in (Brownlee, n.d.)). Partial threading is introduced using the Fork-Join paradigm which is supported by the core Java library in Java 7.

The first opportunity for parallelism, training of each group of B-cells in parallel, is identified by Brownlee (Brownlee, 2005), but not included in the Weka implementation.

The most time consuming part of the algorithm is the affinity calculations across all antibodies and all antigens. Each affinity calculation (the affinity of one antigen to one B-Cell) is independent of all the other affinity calculations. This is a significant opportunity for further parallelism to be introduced. The nested loops running the affinity calculations in series (lines 5-6 of Algorithm 1) can be replaced with a construct that distributes the individual calculations to the available thread pool, and then collects the results as each thread completes. Once all calculations are complete, the normal sequence of the algorithm is resumed.

An example of the fork-join construct is shown in figure 1. The modified algorithm is summarised in Algorithm 2. The fork-join construct is used for the blocks between lines 7-13 and lines 22-30, with the worker threads performing lines 10 and 25 only.

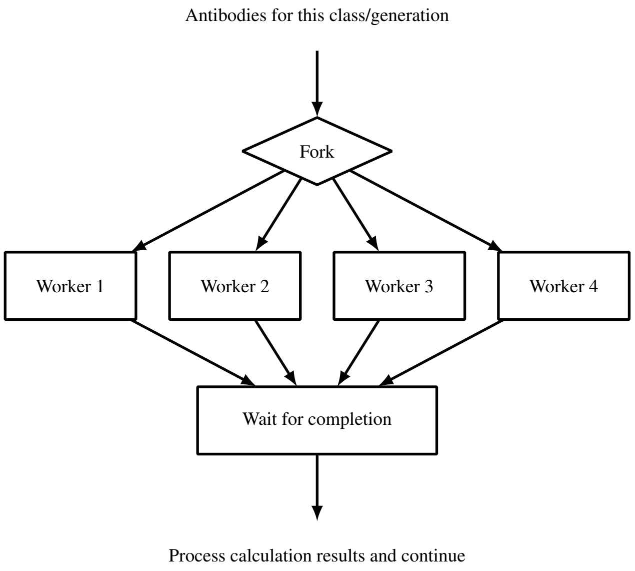
Antibodies for this class/generation
Fork
Worker 1
Worker 2
Worker 3
Worker 4
Wait for completion
Process calculation results and continue

Figure 1: Example fork-Join construct for antibody affinity calculation

The parallelised algorithm uses the same number of operations to train the classifier as the original Immunos 99 algorithm, so the complexity function is the same. If we assume that each computation takes a single time unit, the expected runtime of the modified algorithm is reduced, since affinity calculations occur simultaneously. For the three phases of the algorithm described for the original, and a thread pool of size t, the estimated runtime of the initial setup phase is unchanged (equation 1). The estimated runtime of the modified generation and mutation phase, and of the modified final pruning phase are given by equations 5 and 4, respectively. The runtime of the algorithm is primarily influenced by the number of classes in the training data (since each group of antibodies can be trained in parallel with enough threads) and by the number of execution threads available (reducing the time required to complete the affinity calculations by allowing more to be performed in parallel).

$$g(\frac{a_c + n(\frac{a_c}{t} + (6a_c) + 2(a_c\ log(a_c)))}{min(c,t)}) \tag{4}$$

$$c(a_c + n(2(\frac{a_c}{t}) + a_c + a_c\ log(a_c))) \tag{5}$$

Figure 2 shows the effect of additional threads on the complexity of the algorithm for a hypothetical 4 class problem – using more than c threads has only a small additional benefit. Figure 3 illustrates the effect of increasing the number of classes, where the thread pool size is equal to the number of classes.

Algorithm 2: Parallelised Immunos 99 algorithm

```
1  Create the thread pool, $p$
2  Divide data into antigen groups (by classification label)
3  foreach group do
4      Create initial B-Cell population
5      foreach generation (to parameter limit) do
6          begin Expose all B-cells to all antigens from all
           groups
7              foreach Antigen, $ag$, in the training set do
8                  foreach Antibody, $ab$ in the generation
                   do
9                      begin on a thread from $p$
10                         Calculate the affinity between
                           $ag$ and $ab$
11                     end
12                 end
13             end
14         end
15         Calculate fitness scoring
16         Perform population pruning
17         Perform affinity maturation
18         Insert randomly selected antigens of the same
           group
19     end
20 end
21 begin final pruning for each B-cell population
22     foreach Antigen, $ag$, in the training set do
23         foreach Antibody, $ab$ in the last generation do
24             begin on a thread from $p$
25                 Calculate the affinity between $ag$ and $ab$
26             end
27         end
28         $ab_{best}$ = $ab$ with best affinity to $ag$
29         increment score of $ab_{best}$
30     end
31     Prune antibodies with score below threshold
32 end
33 return the final B-cell population as the classifier
```

Other Parallelisation: the AIRS Approach

Watkins, Timmis, and Boggess report parallelisation of the AIRS algorithm (Watkins, Timmis, & Boggess, 2004; Watkins & Timmis, 2004), which we look to for inspiration. In the AIRS parallelisation the training data is partitioned and sent to thread,s which train independently before merging the resulting memory cells into a single pool for classification. The AIRS approach is more pervasively parallel that

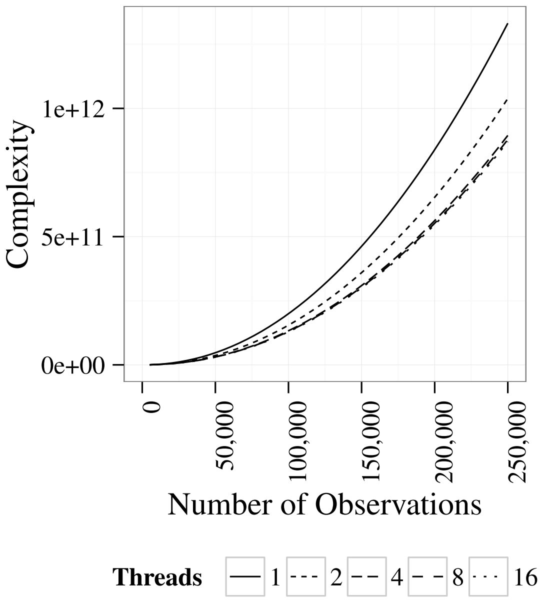

Figure 2: Effect of additional threads on the estimated runtime of modified Immunos 99 on 4 class data.

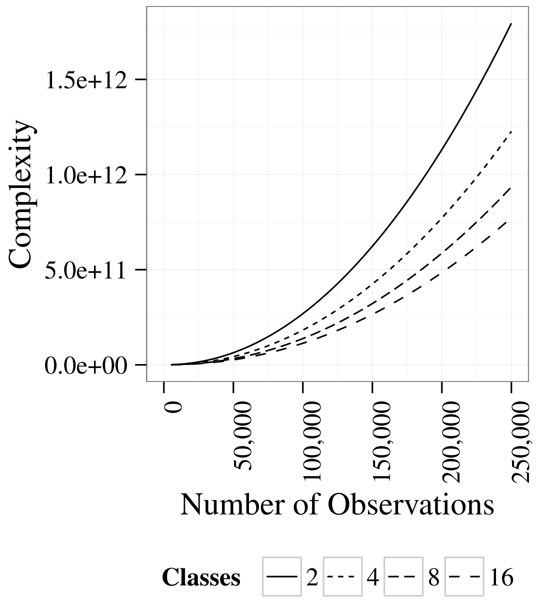

Figure 3: Impact of number of classes on estimated runtime of modified Immunos 99 (with $t = c$).

the Immunos 99 parallelisation, as it distributes large, intensive, independent units of work. In contrast, the Immunos 99 parallelisation approach attempts to gain the largest increase in performance possible without extensive redesign of the original code – the modified Immunos 99 implementation only minimally changes the serial version to distribute the large number of computationally simple affinity calculations.

Reducing Implementation Inefficiencies

The original implementation of the Immunos 99 algorithm is part of the Weka Classification Algorithms project on SourceForge (Brownlee, n.d.). We analysed the execution of the original Weka code using the VisualVM profiler (Oracle Corporation, n.d.). VisualVM identifies the hot spots in the code where improvements are most likely to have an impact. The profiler analysis was used both to guide the algorithmic changes in the previous section and, in conjunction with code reviews, to identify inefficiencies in the use of Java and its libraries. These implementation-specific improvements affect the runtime of the Immunos 99 modified algorithm, but are not factored into the general complexity analysis in Sections and , as they pertain to specific versions of the implementation, Java library and virtual machine.

A significant time overhead in the Weka implementation is the use of array copy operations. For instance, each time the algorithm calls `Collections.sort` from the Java library sorting functions, the collection is copied into a temporary array, sorted, and copied back, rewriting the original collection. We reduce the time to sort a collection by implementing a sort function that directly modifies the array behind each collection without any copying. A simpler modification to reduce copying is achieved by refactoring the Weka code to avoid unnecessary use of methods that rely on copying, such as `Instance.toDoubleArray()`.

The profiler analysis showed extensive use of `LinkedList` objects. `LinkedLists` are inefficient when used in conjunction with random access operations, as in the original code, since every index request requires traversal of the list (average complexity $O(n/2)$). To improve efficiency, the `LinkedList` objects were replaced with `ArrayList` objects, which provide the same interface but have complexity of $O(1)$ access to contents.

A review of the logic of the Weka implementation revealed an inefficiency in the final pruning step (Algorithm 1 line 7). At this step, only the antibody with the best affinity is required, so it is unnecessary to sort all the antibodies by affinity. By replacing a call to `Collections.sort` with a call to `Collections.min`, the complexity is reduced from $O(n\ log(n))$ (for sorting) to $O(n)$ (to scan the collection).

Experimental Results

We illustrate the difference in execution time between the original and modified implementations using the *Chess (King-Rook vs. King-Knight) Data Set* (kr-kk) from the UCI machine learning repository (Frank & Asuncion, 2010). Each test is a 10-fold cross validation of the kr-kk dataset, which is repeated 10 times, with the time for each run recorded. Tests are performed on an otherwise-idle Win-

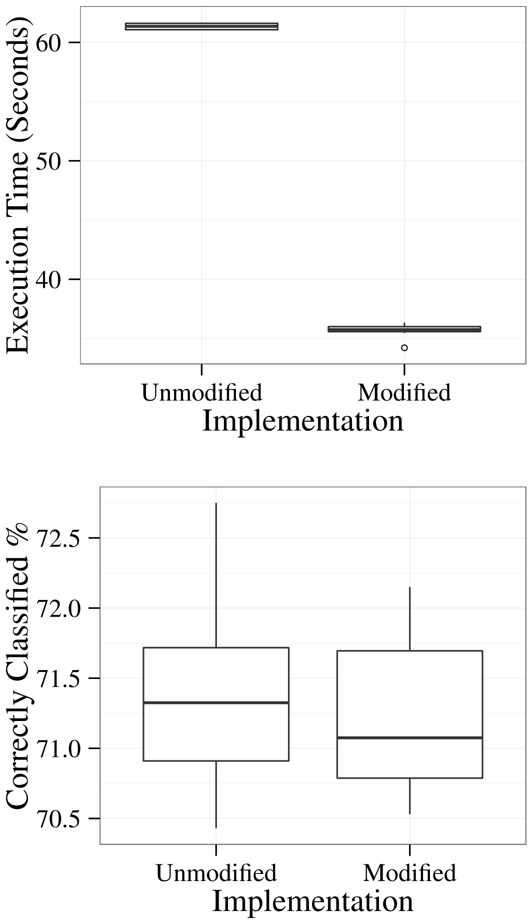

Figure 4: Box plots showing execution times and accuracy for 10-fold cross-validation of each implementation on the chess data set

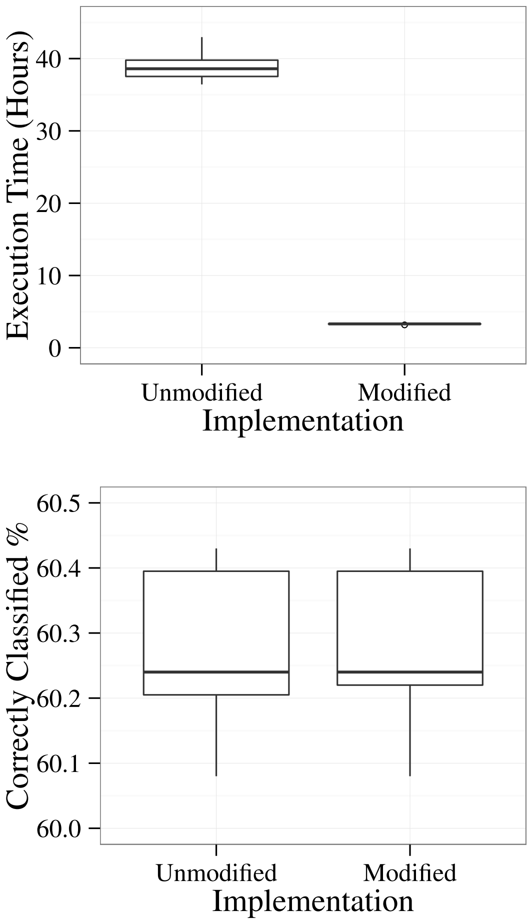

Figure 5: Box plots showing execution times and accuracy for 10-fold cross-validation of each implementation on the proprietary data set

dows Server 2008 box with 16 Cores (4x Intel Xeon E7340 CPUs) and 32GB of RAM. Tests are run under the 64-bit Java 7 JDK (version 1.7.0-b147).

Figure 4 shows the time and classification performance of the implementations. The box plots show the median value as the centre bar; the outer limits of the box are the first and third quartiles, and whiskers show the highest and lowest value within $1.5 * InterquartileRange$ of the box edge. Any outliers are shown as points.

The results for execution time (upper plot in Figure 4) shows that the improved algorithm has a mean execution time that is approximately 58% of the execution time for the original. Both implementations exhibit very low levels of variation in execution time. The outlier in the plot for the modified algorithm is thought to be due to the influence of the operating system scheduler on the server performing the tests.

The lack of overlap between the result ranges of the two implementations indicates a significant improvement in execution time with the modified implementation. A non-parametric Mann-Whitney U test for significance was performed, with H_0 of no significant difference between the performance of the original and modified implementations. The test p-value is 0.0001806, so H_0 is rejected with greater than 99% confidence.

To increase confidence that the classification ability of the algorithm has not been compromised by the modifications, the overall classification accuracy was also recorded for the 10 runs (lower plot in Figure 4). In the results, the modified algorithm has a slightly lower median accuracy, but less variation in classification performance than in the original implementation. Performing the Mann-Whitney U test for significance, with H_0 that there is no significant difference between the performance of each algorithm yields a p-value of 0.6305 so H_0 is accepted.

In the motivation for this work (Section), we describe the training phase execution of the original algorithm on a proprietary dataset, which took in the order of 40 hours to complete. Performance of the modified algorithm was also tested on this proprietary dataset. The experimental setup was identical to that for the kr-kk dataset. The mean execution time was reduced from 38.86744 hours for the unmodified algorithm to 3.29218 hours for the modified algorithm. A Mann Whitney U Test for significance was again performed, with H_0 that the execution time for the two implementations is the same. The p-value of 0.00001083 determines that H_0 is rejected with greater than 99% confidence. The mean accuracy of the algorithms is almost identical: 60.274% and 60.276% for the unmodified and modified implementations, respectively (Mann Whitney U Test p-value of 0.9092). The results (following the same format as before) are plotted in figure 5.

The results are in line with expectation: the quadratic complexity of the Immunos 99 algorithm means that performance gains from the modified implementation should be higher for larger datasets. We have also informally observed increased improvements for larger datasets in other work on proprietary datasets.

Further Work: Proposed Map-Reduce Implementation

In working on the Immunos 99 algorithm, it became apparent that a map-reduce variant of Immunos 99 would support massively-parallel execution. This should lead to feasible training times for very large datasets. The outline is presented here as an opportunity for further work.

Map-reduce (Dean & Ghemawat, 2008; The Apache Software Foundation, n.d.) allows tasks to be broken down into an arrangement of *map* and *reduce* tasks. Map tasks are responsible for splitting the incoming data into chunks that can be independently processed. Reduce tasks are responsible for processing a chunk of data and then returning a result to the next task in the chain. The most time consuming part of the Immunos 99 algorithm (exposing all B-cells to all antigens) could leverage these constructs as follows.

1. Scoring of each antibody within a generation can be performed independently of any other antibody.
2. Each antigen/antibody affinity comparison used to calculate the antibody score can be mapped out as a separate task.
3. Ranking the antibodies by affinity for scoring could be a subsequent reduce task that merges the antibody scores into a sorted list for the calculation (cf. merge sort).
4. Rank scoring of the antibodies for each instance can be done as a reduce task on the sorted antibody scores.
5. Final pruning and other finalising tasks can be performed in serial by a final reduce task.

The final pruning step could be implemented using a similar series of map and reduce tasks, which would further reduce execution time.

The map-reduce approach would allow a refactoring that can efficiently exploit a large number of compute resources to process in a much shorter time than either the original implementation in Section , or the improved variant in Section . A caveat to this is that the time spent administering and distributing the map and reduce tasks over a compute cluster is not insignificant, so for smaller datasets the single machine approach from Section is likely to be faster.

Conclusion

This paper has assessed the practical performance of the Immunos 99 classification algorithm (Brownlee, 2005), considering some performance deficiencies of the algorithm and its publicly-available implementation (Brownlee, n.d.). We present modifications to the existing source code that enable Immunos 99 training to run in an acceptable time (according to the business context) on large, noisy datasets, by effective exploitation of multi-core processors are discussed. We outline additional modifications using the map-reduce paradigm (Dean & Ghemawat, 2008; The Apache Software Foundation, n.d.) which have the potential to support massively-parallel execution.

References

Watkins, A., Timmis, J., & Boggess, L. C. (2004). Artificial Immune Recognition System (AIRS) : An Immune Inspired Supervised Machine Learning Algorithm. *Genetic Programming and Evolvable Machines*, *5*(3), 291–317.

Watkins, A. & Timmis, J. (2004). Exploiting parallelism inherent in AIRS, an artificial immune classifier. In G. Nicosia, V. Cutello, P. J. Bentley, & J. Timmis (Eds.), *Artificial immune systems* (Vol. 3239, pp. 427–438). Lecture Notes in Computer Science. Springer.

Taylor, P., Leida, M., & Majeed, B. (2012). Case study in process mining in a multinational enterprise. In K. Aberer, E. Damiani, & T. Dillon (Eds.), *Data-driven process discovery and analysis* (Vol. 116, pp. 134–153). Lecture Notes in Business Information Processing. Springer Berlin Heidelberg.

Brownlee, J. (2005). *Immunos-81, The Misunderstood Artificial Immune System* (Technical report No. 1-02, Swinburne University of Technology, Austrailia).

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA data mining software: an update. *SIGKDD Explorations*, *11*(1), 10–18. doi:10.1145/1656274.1656278

Brownlee, J. (n.d.). WEKA Classification Algorithms. Retrieved from http://wekaclassalgos.sf.net/

Oracle Corporation. (n.d.). VisualVM. Retrieved from http://visualvm.java.net/

Frank, A. & Asuncion, A. (2010). UCI machine learning repository. Retrieved from http://archive.ics.uci.edu/ml

Dean, J. & Ghemawat, S. (2008, January). MapReduce. *Communications of the ACM*, *51*(1), 107. doi:10.1145/1327452.1327492

The Apache Software Foundation. (n.d.). Apache Hadoop. Retrieved from http://hadoop.apache.org/