A Predictive and Automated Cloud Server Scaling System

ABSTRACT

This paper explores the possibilities of creating software which scales cloud servers pre-emptively as opposed to the traditionally used reactive approach. The objective is to outline the design and implementation of a system which is integrated with a third party cloud hosting service, and can predictively manage a user's server architecture based on the latter's varying needs. At the core of the software, a prediction algorithm is built to classify and associate specific measurements read in a server's environment with server scaling actions. Extensive research is made into the applications of Evolutionary Algorithms in the field of data classification. Following a linear development model, the possible requirements, design and implementation of such a system are described. An experimental and analytical section describes the results of implementing different evolutionary algorithm for the prediction mechanism, in order to determine the most appropriate choice for the final design.

Table Of Contents

1. Introduction

Many businesses and individuals choose to host their websites and applications in the cloud, as it eliminates many of the problems which might be encountered when using physical servers. The increasing amount of users of said applications result in a need for efficient server architectures, which allow software infrastructures to be resilient, yet cost-effective. Cloud hosting providers such as Amazon Web Services, Rackspace and Microsoft Azure have seen a rising growth in demand in recent years. When running applications on a large amount of cloud servers, businesses can be unwilling to spend much of their human resources on server management, and rely on software to accomplish this. Several software solutions exist for the automatic scaling of server architectures, and are usually provided by popular cloud hosting services to their customers, such as Amazon Web Services' Auto-Scaling software. However, these are purely reactive systems, meaning that server infrastructures are only modified after they are required to do so. A pre-emptive server scaling system would identify an infrastructure's different needs at different times, and manage the servers accordingly. Implementing predictive scaling software would theoretically have several benefits:

Reduced cost: Mainstream cloud hosting companies charge their customers based on individual server usage. In addition to this, servers do not take a negligible amount of time to stop and start, and existing auto scaling features might have delayed reactions in carrying out these tasks. This could mean that servers are still running when they might not be needed, which can lead to a significant waste of money. If a piece of software could be designed to only start servers when they are absolutely needed, this would drastically reduce spendings. Furthermore, business personnel which might have previously been assigned the task of monitoring and scaling a server infrastructure can be reassigned to tasks which can not be taken on by automated systems, thus benefiting the business by improving efficiency.

Increased stability: As most web applications experience sudden and drastic changes in traffic, servers hosting these applications are likely to slow down or even crash if a reactive automatic scaling system does not act quickly enough. This problem could be avoided by predicting these changes in traffic.

Essentially, the task at hand can be seen as a classification problem. The program will need to measure abnormalities in groups of virtual servers, and predict a scaling action which will rectify this. The software will need to learn and classify which types of server behaviours require different actions. There are a number of prediction algorithms which can achieve this, some of which will be discussed here. This report will outline the development of such a system using a linear approach, describing the requirements analysis, design, implementation and testing phases of the process.

2. Background Research

Background research for this paper includes some important concepts surrounding cloud servers, and how these oppose traditionally used physical servers. Furthermore, some of the possible design patterns and paradigms which can be employed by such a system are explored. The largest proportion of research details some well known prediction and classification algorithms, particularly Evolutionary Algorithms, although other options are also considered.

2.1 Development Methodology

A Waterfall life cycle (Budgen, 2003) can be adopted for the development of this system. The Waterfall method is a linear approach which covers all areas of software development, from feasibility studies to software implementation. The waterfall model is ideal when building systems which are tailored to a specific user's or businesses' needs. This opposes a more incremental approach such as a spiral model, which is more suited for the development of software which is intended for a community of users, who's requirements and needs for the software might change throughout the process.

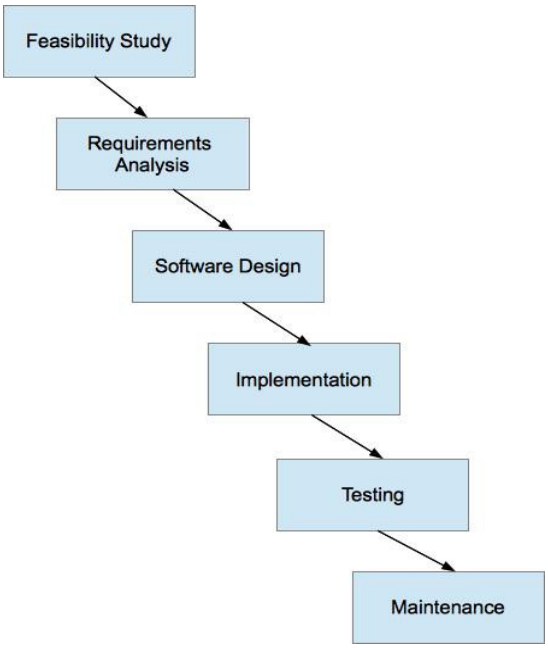


Fig. 1 The Waterfall Development Methodology

The Waterfall development methodology consists of six distinct phases:

- Feasibility study: This process examines whether the software is attainable given the business context
- Requirements analysis: A phase during which the specific needs for the system are gathered, from a user's point of view, in addition to a technical perspective.
- Software Design: The purpose of software design is to detail each aspect of the system, from the high level architectural design to each component and element of the software
- Implementation: Writing the code and creating the data sources which fulfil every system aspect detailed in the design
- Testing: The validation of the system based on the requirements, design and implementation phases
- Maintenance: Almost all software systems require some maintenance. This may sometimes involve an entirely new iteration of the waterfall model.

2.2 Virtual Server Hosting

2.2.1 Physical VS Cloud Servers

Cloud computing can be defined as as "A set of Internet-based application, storage and computing services sufficient to support most user's needs, thus enabling them to largely or totally dispense with local data storage and application software" (Coulouris, 2012) Essentially this would imply that users of cloud servers are able to host large, resource intensive applications without the use of physical, dedicated servers. Cloud technology means that businesses can avoid many of the burdens which were previously encountered when using physical servers, such as large facilities for their storage, ventilation systems and hardware technicians. Virtual servers provide many other benefits (Abidi, Singh 2013):

- Flexibility: Cloud servers can be created, modified and deleted with a few clicks or an API call to the service provider.
- Stability: Cloud servers have a significantly reduced risk of hardware malfunction, when compared to physical servers
- Cost: Hosting applications in the cloud is no longer considered a novelty; In most cases buying more dedicated servers will be more costly than creating new cloud server instances. Instead of paying a fixed price, users only pay for what they use.
- Reduced need for technical ability: Hosting applications in the cloud does not require much knowledge of server hardware, and business personnel will never be required to repair machines in

case of failures. Users simply need to be familiar with the User Interface provided by the hosting service for server administration, and have some basic technical knowledge.

- Cloud servers can be accessed and modified from anywhere using Secure Shell (SSH) technology, or through a graphical remote login.

Cloud servers can be hosted by third parties, such as Amazon Web Services and Rackspace, or through an internal infrastructure. These are often referred to as being virtual, as unlike physical servers, they are not hosted by and limited to a single machine.

2.2.2 Virtual Server Hosting : Important Concepts

The following aims to define some important concepts surrounding cloud computing, which will be referred to extensively throughout this report.

- A server cluster (Coulouris, 2012) is defined as a a network of various computers which interact and cooperate to form a single computational infrastructure
- Scaling is the process of modifying the current server cluster. This involves either horizontal scaling, which is the process of adding or deleting more servers to the infrastructure, or vertical scaling, which allows pre-existing servers to be modified by changing their resources such as CPU and memory.
- Creating a cloud server requires hardware configuration (Norton, 2002), such as specifications for the RAM and CPU capacity and the server's Disk space.

Servers can have different purposes:

- Web servers: Used to host the front end (User Interface), or web services of an application, and provide the main entry point to an infrastructure.
- Application servers: Host the business application.
- Database servers: Used to host an application data source
- Load balancers: A server which is dedicated to hosting load balancing software, which distributes traffic between members of the cluster.
- Spreading system components across separate servers as such has the advantage of maintaining and isolating any eventual malfunctions, reducing the possibilities of system wide failures.

2.2.3 Scaling Horizontally VS Scaling Vertically.

When dealing with scalable applications which are hosted in the cloud, it is important to have a clear idea of the differences between horizontal and vertical scaling (Micheal et al., 2007). Neither are suitable for every situation, and it is important to understand when one option needs to be preferred to another. Vertical scaling refers to the modification of pre-existing servers, whereas horizontal scaling implies the addition or removal of servers from the cluster. Many distinctions between vertical and horizontal scaling have been made:

- Vertical Scaling is suitable for use when an application is being bottlenecked by a specific resource, as an administrator can simply increase the resource in question as required
- Horizontal scaling is usually employed once current servers have been heavily optimised and are stable enough to be replicated
- As networks become increasingly unstable as they grow, excessive horizontal scaling is to be avoided unless vertical scaling becomes inefficient
- Vertical scaling is limited to the maximum amount of hardware resources provided by a cloud hosting service per server.

2.2.4 Cloud Hosting Service Providers

Many businesses offer cloud server hosting services, the most popular of these include Amazon Web Services (Amazon, 2014), Rackspace (Rackspace, 2014) and Microsoft Azure (Microsoft, 2014). These services all follow a similar business model, and allow virtual servers to be created in a comparable fashion:

- Virtual servers can be created and booted with ease, these are usually referred to as Virtual Machines (VM)
- Virtual Machines are based on machine images, which essentially represent the Operating System to be installed on a server. All three services support both Windows and Unix based operating systems, and allow their users to to load custom machine images, which can contain any additional software a user might require.
- A loaded machine image which is assigned a configuration of server resources is known as a server instance. Although these providers do not allow their users to specify every exact resource allocation for their instances, they offer a large variety of instance configurations for different application needs.
- In all three cases, payments are based on duration of server uptime, and the number of servers in use.

- The three services offer extensive public APIs which covers many programming languages, meaning that it is possible to design and implement software which integrate with the services.

2.2.5 Scaling Criteria

Existing predictive server scaling applications, such as Netflix's Scryer (Netflix, 2013) software, base their scaling actions on current server resources: CPU and RAM usage, Disk Write speed and current application usage. It is also usually possible for users to configure such applications manually to allow scaling actions to be taken at specific times of days. However, we can identify additional metrics which can cause a server's behaviour to change:

- Reoccurring or severe application exceptions and errors
- Traffic spikes
- Database errors and time-outs
- Network errors
- Poorly written application code
- Issues with another server, resulting in large amounts of traffic being redirected by a load balancer

2.3 Management Science

Due to the nature of the proposed system, scaling predictions might not be completely satisfactory and reliable. This is particularly pertinent if the prediction algorithm's learning process is not optimal, and needs certain parameter changes. This could cause considerable damage to any business looking to adopt the system, as servers could potentially be started and stopped in the wrong situation if the software is left to work completely autonomously. A case like this could lead to a considerable waste of money for unnecessary servers, or downtime of a business application if too few servers exist within a cluster to support it. Our software could therefore be used as a Decision Support System, which could give recommendations to experts as to which actions might be suitable for specific cases. The software would run in such a manner until experts are entirely satisfied with its output; at this point it would be made to run autonomously, and left in charge with the decision making process.

Management Information Systems and Decision Support Systems are two distinguishable types of systems in management science (Finlay, 1994):

- Management Information Systems (MIS): These are built in situations where decisions and actions can be predefined by experts, and have the aim of automating certain business processes and replacing human personnel .
- Decision Support Systems (DSS): In contrast with the former type of system, these are employed in situations where software can act as a decision recommendation service, but where human expertise is still essential. These systems do not make any decisions autonomously, but help in influencing an expert's decision making process.

Once implemented, our software could be used as each type of these management systems, in two separate phases:

- In a first phase, the software could be employed as a DSS, as too much risk would be involved in allowing it to run autonomously, should wrong decisions be made. It could therefore be employed as a support system for Server Administrators, recommending decisions for starting, stopping or modifying servers. The administrators could detect any recommendations which they deem wrong, and request that the software's development team make source code changes.
- Once the administrators are entirely satisfied that the software's decisions are guaranteed to be correct, the scaling software can be refactored to an MIS, by integrating it with the business application's server cluster, allowing it to fully make cluster modifications. This would correspond to a maintenance phase of the Waterfall model.

2.4 Software Design Patterns

2.4.1 The Client/Server Design Architecture

It is important to design these types of systems such that functionality is distributed across several components of the architectural structure. Several key benefits to designing systems in a distributed fashion have been outlined (Coulouris, 2012).

- Systems and applications can share resources and function concurrently and simultaneously.
- Failures are maintained within the system. If a single component fails, which is expected to occur occasionally with all software, the failure can be more easily maintained within the component which is affected, making it less likely to bring down other parts of the system.

The Client/Server architectural paradigm has been used extensively, due to its simplicity and effectiveness. Typically, a client will make a request to a server, demanding or supplying specific resources or data, and the server will proceed with an action, supplying the client with a result.

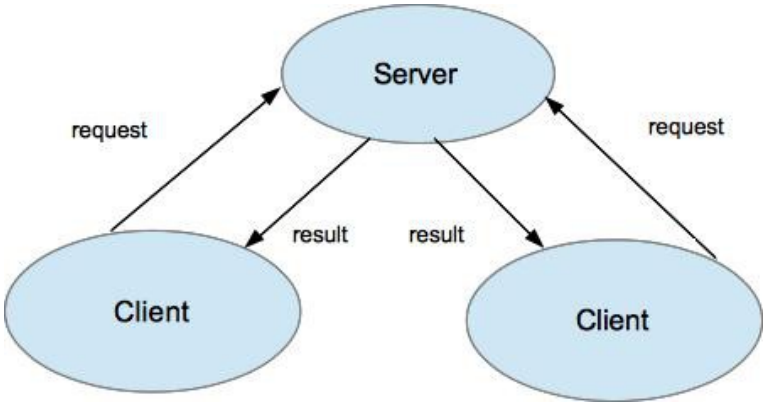


Fig. 2 Diagram of the Client-Server Software Architecture

Systems which are designed following the client/server paradigm contain a single Server, which communicates with several Clients, exchanging resources and data. An important consequence of adopting the client/server model is the ease at which these types of systems can be scaled (Crichlow, 2000). The model also allows for the separation of labour across the system, and is the most easily implementable distributed system, as opposed to a distributed model such as P2P.

2.4.2 The Service Oriented Architecture

The Service Oriented Architecture (SOA) paradigm describes how large object-oriented systems can be separated into multiple tiers, each with it's own purpose. These layers are separated based on their role in the system, and consecutive layers are interconnected (Fowler, 2003)

- The presentation layer contains the display logic for the User Interface, or web services, and makes requests to the application itself.
- The service layer contains services which encapsulate domain layer logic, and executes this logic in such a way which fulfils specific system functionality. Services can be executed in many ways such as:
  - Through the presentation layer by a user
  - Triggered by incoming data to the system, through the web services
  - Through autonomous and regular system behaviour.
- The domain layer contains the business logic of a system. In an object oriented system, this will take the form of classes which represent concepts and objects.
- The mapper layer provides the link between the data source and the domain logic. This usually takes the form of dynamic database queries which have the role of selecting, adding, modifying and deleting data from one or multiple databases
- The data source contains the database, which drives the domain layer by populating and saving it's objects.

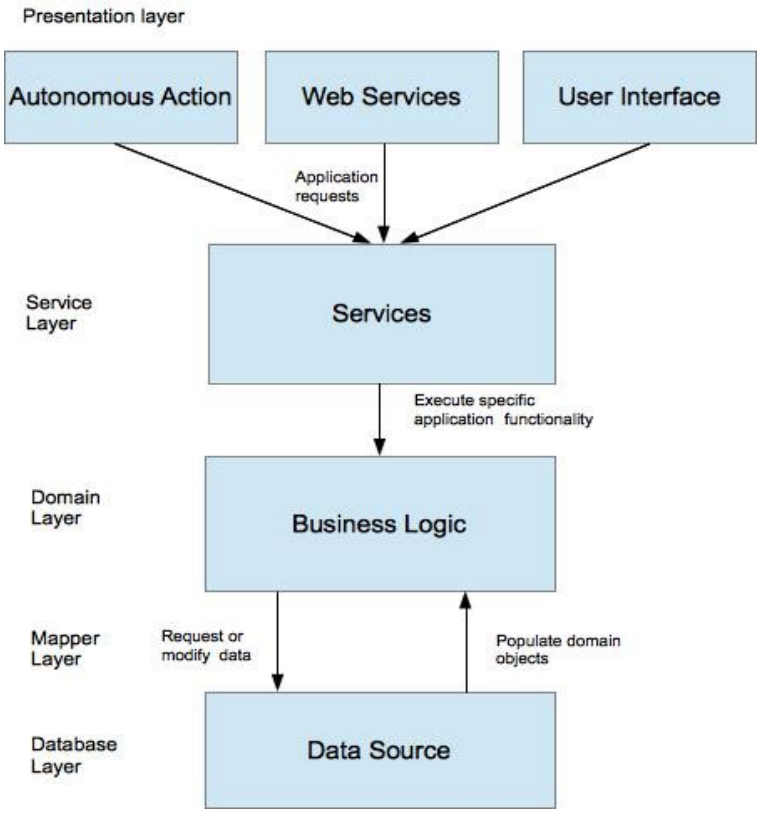


Fig. 3 The Service Oriented Architecture

2.5 Prediction and Classification Algorithms

The task of classifying scaling actions can be seen as non linearly separable (Hastie, 2001). To explain this, one must consider a two dimensional model of the search space. If a single distinct line can separate the elements of the space based on their classification, the problem is considered as linearly separable. It is sensible to assume that this is not the case for the classification of server scaling actions, as a very slight change in a server metric can cause the need for a distinct scaling action. A very basic example of a problem which is not linearly separable is the exclusive OR problem.

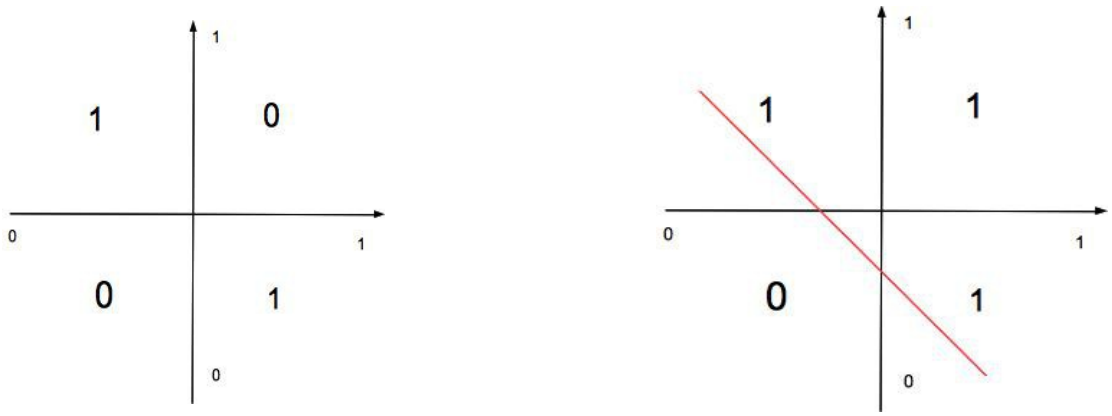


Fig. 4 Linear separability of the XOR (left) and OR(right) problems

There are a number of machine learning algorithms which are capable of solving non linearly separable problems, some of which are discussed in the following section.

2.5.1 Naive Bayes Classifiers.

The Naive Bayes Classifier is a statistical method for determining the probability of an element being associated as a specific class (Hastie, 2001). The naïve Bayes classifier is a form of supervised learning, based on Bayes theory, which dictates that the probability of an event A occurring given B is calculated as such:

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$$

The method is named 'Naive' as it assumes that features of an element to be classified are independent, and it is presumed that combinations of features to not collectively contribute to classification. Naive Bayes classifiers utilise this formula to determine the odds of an element's classification. Although feature independence is assumed, the Naive Bayes has proven successful in solving a number of problems. Recent research (Maruyuma, 2013) successfully classified complex protein structures using a Bayesian approach.

2.5.2 Neural networks

A neural network is a machine learning and classification technique which draws inspiration from biological events observed in nervous systems (Patterson, 1996; Picton, 2000). Neural networks are designed and trained to be presented with an input, and produce a corresponding output. These types of classification algorithms are capable of solving non linearly separable problems meaning Neural Networks are a viable choice for predictive software. As this is a form of Machine Learning, neural networks are designed in such a way which allows them to adapt and evolve based on varying input patterns. The perceptron is the most commonly used variation of the Neural Network, and consists of multiple layers of interconnected nodes. Two values are associated with each node: A weight parameter and a threshold parameter. The output of a node is calculated by multiplying the nodes weight value by the input. If the value of this calculation is above the threshold, a certain value is output by the node, otherwise a different value is output. Multilayered Perceptrons can use a back propagation algorithm, allowing the weights and thresholds of nodes to be adjusted based on error detection, meaning the algorithm can adapt autonomously. Neural networks are widely used in the field of Machine Learning, and have proven successful in many cases. A recent study (Azar, El-Said, 2013) successfully classified benign and malign tumours based on characteristics of certain female patients suspected of having breast cancer.

2.5.3 Genetic Algorithms

Although work on algorithms inspired by evolution began as early as the 1950s, Holland coined the term 'Genetic Algorithm' in the 1970s (Holland, 1973). Genetic Algorithms are traditionally used in optimisation problems, and are loosely inspired by evolution, and Herbert Spencer's theory of “survival of the fittest”. The algorithm is based on a simple process which is repeated until termination criteria has been satisfied, such as a specific number of iterations, or an optimal solution has been found. The process is as follows: A population of individuals are made of genes which represent potential solutions to the given problem. Individuals are evaluated based on their aptitude at solving the problem, and assigned a fitness value. The fitter individuals of the population are then selected for reproduction. The higher an individual's fitness, the

more likely it is to be selected for this phase. In essence, the algorithm works around the fact that by combining the genetic material of two fit individuals, the result will often be of even better quality. The crossover operator mixes the genes of pairs of individuals, in order to keep a certain degree of diversity within the population. However, it is through mutation that the learning process will fundamentally be achieved, as it allows new areas of the search space to be explored through random genetic modifications Following the application of the genetic operators, the new generation of individuals replace their predecessors, and the entire process is repeated, until the termination criteria has been achieved.

2.6 Evolutionary Algorithms

For the sake of this classification problem, we have chosen to explore the route of using Evolutionary Algorithms. Although these have traditionally been used for optimisation problems, we can draw inspiration from the Learning Classifier System to use Evolutionary Algorithms as a classification tool. The latter is a method which uses a Genetic Algorithm at it's core in order to solve classification and prediction problems. The term Evolutionary Algorithm is used to refer to the subset of machine learning algorithms which are based on theories of evolution, of which the Genetic Algorithm is a member. The following provides and in-depth discussion of three different Evolutionary Algorithms: The Genetic Algorithm and the Baldwinian and Lamarckian Memetic Algorithms.

2.6.1 The Original Genetic Algorithm

As previously described, Holland's Genetic Algorithm exploits the concept of a species' ability to adapt throughout it's generations in order to survive. The basic algorithm process is as follows (Eiben, 2003):

Initial population generation: The population consists of a specific amount of individuals, each made of genes which are randomly assigned, and represent potential solutions to the task at hand.

Fitness evaluation: Individuals are evaluated based on their performance at solving the problem. Individuals with greater fitnesses represent more suitable solutions.

The selection process: The purpose of this mechanism is to ensure that mostly fit individuals are selected to be reproduced. However this needs to be somewhat lenient, as repeatedly reselecting similar individuals must be avoided as to allow the population to maintain a certain degree of variety. For this reason the selection process must be somewhat stochastic. Tournament selection involves randomly selecting a subset of the population (the tournament) based an a user defined tournament size. The fittest individual of the tournament is returned as the selected member. Roulette wheel selection is a simpler and less efficient method by which the odds of an individual being selected are directly proportionate to it's fitness.

The crossover operator: The process by which pairs of individuals combine their genetic material to form new children solutions. This mechanism is loosely based on biological reproduction. Several crossover

methods exist, however the uniform crossover is the most widely used, as it allows more varied individuals to be formed. The concept of uniform crossover is straightforward: Each gene of an individual has a probability of crossing over with the gene at the same position of it's pair, based on a user defined crossover rate. Single point crossover randomly selects a gene index, and performs a crossover between two individuals for every gene past the selected index. The following diagram shows the likely effects of uniform crossover with a rate of 50%, using individuals of ten genes

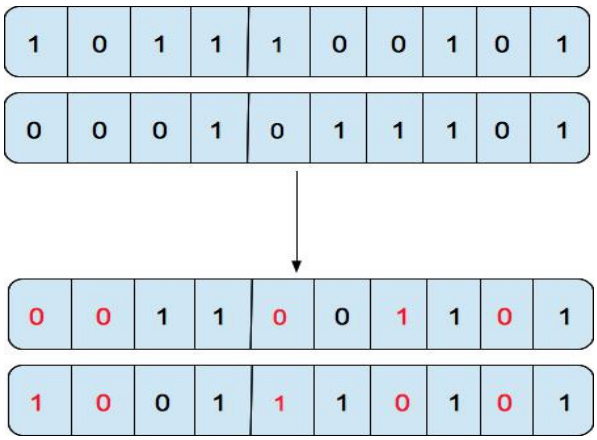


Fig. 5 Visual Representation of the Uniform Crossover Operator

- **The mutation operator:** This is loosely based on the rare biological phenomenon of genetic mutation, a process by which a foetus spontaneously modifies a portion of it's genetic material post-conception. In computational terms, this corresponds to simply modifying a single gene to another possible genetic value; Each gene has a probability of mutating, although the mutation rate should be significantly low. The following diagram shows the likely effects of the mutation operator with a 0.1 probability, using individuals of ten genes

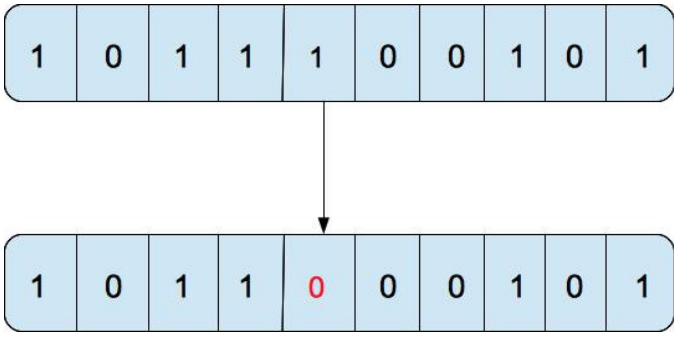


Fig. 6 Visual Representation of the Mutation Operator

Elitism: This concept allows the learning process to remain progressive, by always ensuring that the fittest individual of a generation is reselected for the next generation. Implementing elitism means the best fitness of the population over the iterations of the algorithm is never in decline, allowing for a more rapid and linear learning rate.

The process is repeated until termination criteria has been satisfied, which is usually when the best possible solution has been found, or a maximum number of iterations of the algorithm have taken place.

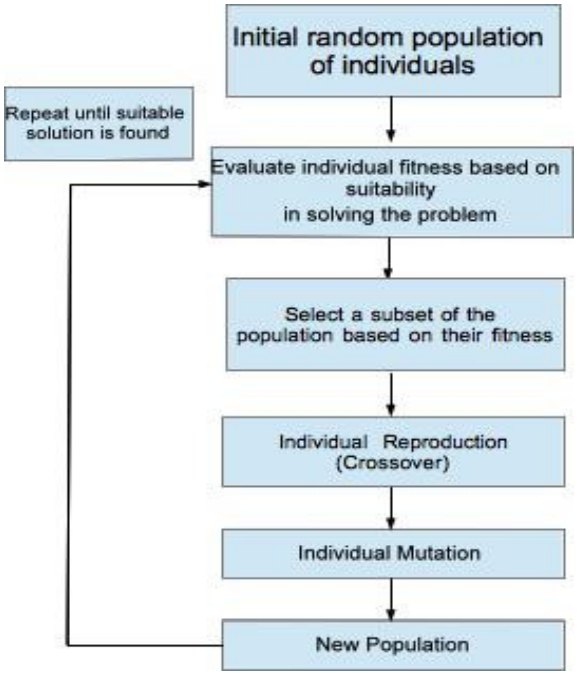


Fig. 7 Logical Flow Diagram of the Genetic Algorithm

2.6.2 The fitness function

The fitness function is the most important feature of the Genetic Algorithm, as it distinguishes high quality solutions from lower quality solutions. Two fitness evaluation techniques can be considered for this application (Petridis et al., 1998):

- A penalty based system, by which individuals have an initially large fitness value, but see this value penalised for being too similar or equal to user defined constraints.
- A reward based system, by which individuals have an initial fitness of 0, but are awarded fitness increases for being similar or equal to user defined objectives.

2.6.3 Holland's Learning Classifier System

The Learning Classifier System was developed by Holland (Holland, 1976), and is an ideal system for tasks which involve element classification and prediction. It combines both elements of reinforcement and evolutionary learning. The LCS would be suitable for the given problem, as it is capable of generating rules which cover an entire input space and provide the most appropriate action. Rules are made of condition (input) segments, and action segments (output) In a non static environment, the LCS is capable of modifying and adapting the generated rule base as required. The LCS is a reward based algorithm, which incorporates a Genetic Algorithm, and operates in two phases (Bull, 2005):

- The rule evaluation phase: The condition portions of rules in the rule base are compared to the input received from the environment in order to select an appropriate rule. The rules which match are added to the collection of possible rules for the current cycle, known as the match set. The corresponding actions are grouped based on their action type, and added to subset of potential actions known as the action set. Predicted pay-offs (rewards) are calculated for each action, and an action is selected based on these calculations. Although this mechanism varies, a simple approach would be to select the action with the highest predicted payoff. The action is then carried out, and rewards are allocated to the rules in question.
- The rule discovery phase: A Genetic Algorithm is run to discover new rules. Individuals represent rules, and the fitness of the parents corresponds to their previously allocated predicted pay-off. Children are created through the mechanisms which have been previously described, and their fitness can be calculated by averaging the fitness of their parents. The Genetic Algorithm has a regular probability of running, in order to adapt the rule base to a changing environment.

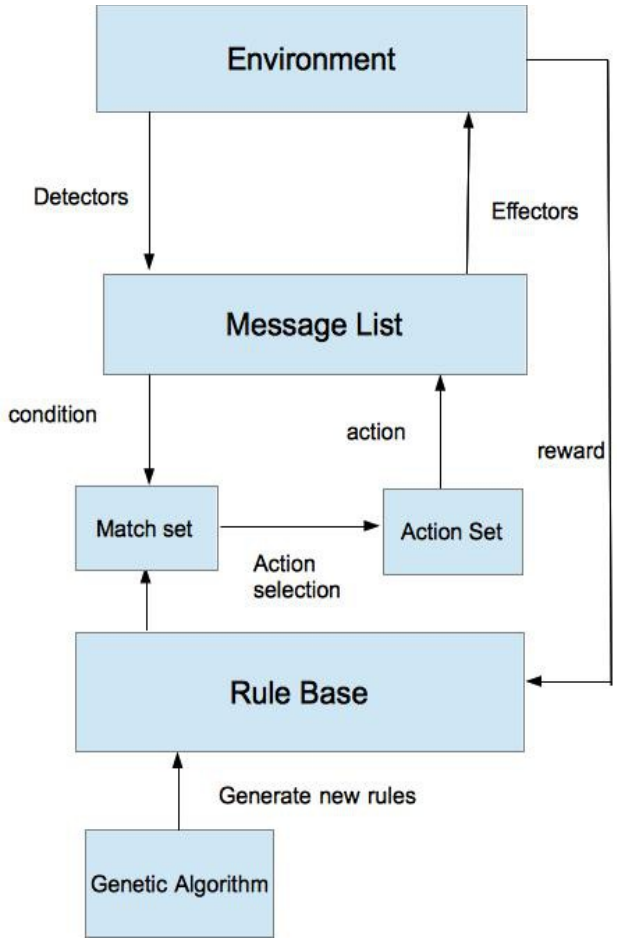


Fig. 8 Logical Flow Diagram of the Learning Classifier System

LCS logical diagram description:

- Input is detected from the environment via the message list
- Using the input, all suitable rules in the rule base with the appropriate condition portions are added to the match set.
- The rules in the match set are grouped based on their actions.
- Every rule of the match set which contains an appropriate action is added to the action set
- A single action is chosen based on a variable selection function which involves the predicted payoffs of the different possible actions.
- The action is then effected, and a reward is received from the environment which is distributed amongst the rules of the action set.
- The Genetic Algorithm is run regularly to explore new solutions which are added to the rule base, and delete unused rules.

- The message list can contain information about previous inputs and actions, essentially giving the system a ' memory'

The success of the LCS lies in it's ability to make generalisations about input received from it's environment. These represent similarities between different environment inputs, and allow rules to be formed about common environment behaviour. Usually a generalisation takes the form of a '#' found in condition portions of rules.

Although we will not be implementing a Learning Classifier System in this report, it is an extremely viable means for carrying out the required task, due to it's versatility. The idea of using an Evolutionary Algorithm to evolve a rule base of generalised if:then rules will be used as inspiration throughout this paper.

2.6.4 Solution Representation

Choosing an efficient and consistent representation of the problem space is vital when employing evolutionary algorithms. An inefficient genotypic representation for solutions can mean that solutions which are strong on the genetic level equate to weak behaviour, meaning that the algorithm is lead in the wrong direction. An unsuitable representation can also lead to the algorithm coming to an end solution which might be highly unsuitable or even infeasible for the given problem.

Representation strategies

The concept of using a Genetic Algorithm to generate a rule base within a Learning Classifier System lead to two distinct individual representation strategies (Ishibuchi et al., 1997):

1. The Michigan strategy: A single solution represents one potential rule for the rule base.
2. The Pittsburgh strategy: A single individual consists of an entire potential rule base.

Binary Representations

The use of a Binary encoding scheme is a widely used technique when designing solution representation for Evolutionary Algorithms. Although real-valued representations are possible, binary representations offer several benefits. These types of representations have the advantage of allowing the genetic operators to be easily applied to individuals. Provided the encoding scheme is designed consistently, an evolutionary

algorithm using a binary representation scheme should yield a significant learning rate.

Certain guidelines must be followed when using encoded values for solution representation (Rothlauf, 1991):

- "Encodings should be adjusted to a set of genetic operators in a way that the building blocks are preserved from the parents to the offspring" (Rothlauf, 1991)This essentially means that desirable portions of the binary strings, which contribute to a higher fitness of an individual, should be kept through the consecutive generations with a suitable representation .
- Solution representation should limit epistasis. This builds on the previous point; Epistasis can be defined as being the possibility of when several parts of a binary string rely on each other to contribute towards the high fitness of an individual. Applying genetic operators to these fractions could potentially destroy desirable binary combinations.
- Useful representations should also be kept as short as possible (although without sacrifice)
- Representations should ensure a high locality, which is the level at which neighbouring genotypes correspond to neighbouring phenotypes. A widely used method for ensuring high locality is a Gray Encoding

Gray Encoding

A drawback of employing binary representations is that the Hamming distance between to consecutive binary strings varies. This means that a single change in the binary representation of a solution can result in a significantly larger change in the solution that the representation describes. Gray encodings can be employed to avoid this (Chakraborty, Janikow, 2003). The Gray algorithm is designed in such a way that two consecutive decimal values correspond to two consecutive Gray values, as shown in the following table. The values highlighted in red represent the changes from the previous value.

DECIMAL	GRAY	BINARY
1	001	001
2	011	010
3	010	011
4	110	100
5	111	101

Using Gray encoding should allow the algorithm to sample the search space in an incremental fashion,

meaning that is is likely that most solutions will be evaluated. Furthermore, it avoids sudden changes in solutions through mutations, which should provide a more progressive and efficient learning process.

Pseudo code for Gray Encoding:

```
Add first bit in Binary String to Gray code
For Each bit in Binary String
        XOR current bit with next bit
        Add the result to the Gray Code
End For
```

2.6.5 Constraint handling

An issue with using binary representations is that it allows for infeasible solutions. Within this context, a constraint can be defined as a solution which is genetically correct given the representation scheme, but is not an encoding of any desirable or even feasible behaviour.

Most problems which can be solved using Evolutionary Algorithms have constrained environments, meaning that the search space of potential solutions is disjointed, and contains regions of feasible and infeasible solutions. This phenomenon occurs due to the presence of potential solutions which are genetically valid, but which map to behaviour which is infeasible within the scope of the given task.

Four different methods of handling constraints in Genetic Algorithms can be outlined (Eiben, 2003):

- Using penalty functions which exponentially reduce the chances of an infeasible solution being reselected over future generations. The penalties are applied preferably “so that the fitness is reduced in proportion to the number of constraints violated, or to the distance from the feasible region” (Eiben, 2003). This is known as indirect constraint handling
- Slightly modifying infeasible solutions to form feasible ones which are not distant in genetic structure.
- Using heuristics to ensure that infeasible solutions are never generated through crossover and mutation, or in the initial random population. This technique in addition to the former are known as direct constraint handling methods through use of enforcement.
- Managing the mapping from genotype to phenotype in such a way that infeasible solutions are decoded into feasible solutions.

2.6.6 Learning issues

In this specific case, it is quite likely that potentially fit solutions exists close to a constraint, but are not being explored by the Genetic Algorithm for this very reason. Furthermore the Genetic Algorithm can become inefficient if the search space is not linear, and are likely to converge towards local optima, particularly if the latter is in the vicinity of solutions of inferior quality (Goldberg, 1998). In addition to this, the algorithm is not always successful in fine tuning solutions which are close to being optimal (Garg, 2009).

There are a number of algorithms which have been developed, and proven to succeed in solving the issues described. These algorithms employ a traditional Genetic Algorithm at their core, but add extra mechanisms for exploring solutions which were very unlikely to have been otherwise.

2.6.7 Memetic Algorithms

Memetic Algorithms (Hart et al., 2005) operate in a similar fashion to the Genetic Algorithm. However, rather than referring to the behavioural encodings as genes, they are known as memes, as the Memetic Algorithm utilizes the notion of cultural learning as opposed to genetic learning. Memes are cultural artefacts which are transmitted from parents to children through the process of imitation, rather than reproduction. Memetic Algorithms also introduce the concept of an individual's ability to adapt to it's environment, which translates to computing a local search algorithm in an attempt to improve the individual's fitness. This takes place directly following an individual's evaluation. For certain problems, Memetic Algorithms can provide a significant learning improvement over Genetic Algorithms (Smith, 2005). The functionalities of two Memetic Algorithms are explored here, based on concepts known as the Lamarckian and Baldwinian effects

The Darwinian Algorithm

The term Darwinian Algorithm will be used to refer to the Genetic Algorithm as previously described.

The Lamarckian Memetic Algorithm

The Lamarckian Algorithm (Hart et al., 2005) is a Memetic Algorithm which contrasts with the traditional Genetic Algorithm, as it makes use of a different theory of evolution which opposes Darwin's theory. The basic mechanisms are the same as the traditional Darwinian Algorithm: Individuals are evaluated, the fittest are selected for imitation and individuals are recombined and inherit characteristics from their parents.

However, Lamarck believed that adaptation changed individual gene encodings to reflect this, and adapted genes could be passed on to future generations. Although this concept is not biologically correct, it's application can potentially result in improved learning, as it implies that an optimal solution can be reached within much fewer iterations in comparison to a Genetic Algorithm. Individuals which are adapted are more likely to have more offspring than those which are not. This is achieved implicitly, as adapted individuals are likely to have a higher fitness compared to the average fitness of the population, increasing their chance of being reselected several times throughout a single selection process.

The Baldwinian Algorithm.

The Baldwinian algorithm is similar to the Lamarckian approach, as it the algorithm also allows individuals to adapt to their environment through experience. It introduces the notion of plasticity, which is defined as an individual's adaptation potential. The higher an individual's plasticity, the more likely it is to adapt (Gançarski, Blansché, 2008). Unlike Lamarckian theory, once the adaptation and selection process has ended, the individuals are reproduced using their initial material, and not the characteristics which were gained as a result of adaptation. This means that the individuals which are predisposed to having a high adaptation rate tend to be reselected in future iterations of the algorithm. This contrasts with the Lamarckian algorithm in two different ways:

- Not every individual of a population will adapt throughout it's lifetime. This is dependant on it's plasticity rate. The plasticity of an individual is encoded as part of the individual's memes, and is a factor in the evolutionary process; It is initially randomly assigned, and is subject to mutation and crossover.
- The behaviour which individual's might have gained through adaptation during their lifetime are not passed on to their children.

The Lamarckian and Baldwinian algorithms are each suited to different types of problems (Khengab et al., 2012). It has been shown that the Lamarckian algorithm is most suitable when a rapid convergence towards a single solution is required, whereas the Baldwinian algorithm thrives when large sampling of the search space is required. The Baldwinian algorithm can also be successful in finding high quality solutions which neighbour low quality solutions; An individual with a low fitness but a high plasticity rate is likely to find a neighbouring solution of greater quality through adaptation.

2.6.8 Local Search Algorithms

As Memetic Algorithms employ local search algorithms for individual adaptation, it is necessary to outline some of the available options for their implementation. Local search algorithms examine problem solutions which exist within close range of the current solution, in an attempt to find improvements.

Hill Climbing

Hill Climbing is a common local search algorithm which can be used by both the Lamarckian and Baldwinian algorithms (Wang et al., 2009). This has commonly taken the form of a single Hill Climbing operation: Each neighbour of the current solution is evaluated until a fitter solution is found, which replaces the current solution (Emile, 2003). This is repeated until an optimal solution has been reached. If a higher quality solution is not found, the individual's genes and fitness remain in their current state, and the algorithm terminates. Hill Climbers have a tendency to prematurely end the search process when a local optima has been found, as only paths which lead to solutions of higher quality are considered. In a non linear search space this is likely to result in a local optima being accepted as the best possible solution of the current solution's neighbouring environment, even though a globally better solution might exist.

Simulated Annealing

Simulated annealing (Van Laarhoven, 1987) is a local search algorithm which has the advantage of being more efficient than the Hill Climbing algorithm at finding global optimal solutions. The process draws inspiration from annealing processes, whereby metals are heated suddenly, and gradually cooled down. The algorithm works in a similar fashion to the hill climber: Neighbours of the current solution are examined, in an attempt to find a solution of higher quality. However, the contrast lies in the algorithms ability to accept inferior solutions at a particular state. As most search spaces are not linear, it is often necessary for the algorithm to take a path which leads to an inferior solution, in the hope that the latter might be neighbouring higher quality solutions. The notion of temperature is an important factor in the simulation process; The higher the temperature, the more likely the algorithm is to examine areas of the search space with inferior solutions, in order to potentially find greater ones. In the initial iterations of the algorithm, the temperature is high so as to sample larger areas of the search space. As the algorithm progresses, and the temperature descends, less solutions of inferior quality to the current solution are examined. Simulated annealing can yield significant results when used as the local search method in a memetic algorithm (Digalakis, Margaritis, 2004).

Following initial research into the problem domain, it is now possible to gather requirements for the software, based on the previously acquired knowledge.

3. Software Requirements

Requirements analysis is the first phase of the waterfall development method, once a system's proof of concept and feasibility have been established. Gathering a list of comprehensive requirements is essential for any software system to be designed successfully. Requirements are used to gather specific system needs from the perspective of every project stakeholder, in addition to a technical perspective.

3.1 Structured Requirements Analysis

A structured analysis requirements gathering approach (Grady, 2006) classifies each requirement into sub-requirements. Employing a structured approach allows requirements to be gathered for all aspects of a system, and not simply those which are visible from the users perspective. By breaking down the problem into smaller sub-problems, we can describe the individual requirements with great detail in order to form a cohesive and coherent view of the various system necessities. The different sub requirement categories for the structured analysis approach are:

- Architectural
- Environmental
- Functional
- Design
- Structural
- Behavioural
- Performance

3.1.1 System Requirements

Functional Requirements:

These can be examined from a user's perspective. As seen in appendix A, a diagram which describes the system's global behaviour, the following functionality must be achieved:

- Training a prediction algorithm to classify server scaling actions based on server measurements. This

must occur periodically, as the environment is assumed to be non static, and the predictions must evolve depending on user defined scaling requirements.

- Performing specific actions based on metrics read from the software's environment – the server cluster. These metrics are compared to the rules previously generated, in order to determine the correct action to be taken.

Architectural Requirements:

The system as a whole will be based on the Client/Server architectural paradigm. This will be beneficial to our system, as many machines will need continuous and concurrent access to the prediction algorithm :

- The sever component will run on a dedicated server, and will have the role of modifying the server cluster as required, based on input received from the clients.
- The client component on the other hand will run on every server which is instantiated with the intention of hosting the business application.
- Both the client and server will be based built on the Service Oriented Architecture paradigm, as the emphasis will need to be put on running services regularly.

Environmental Requirements:

The software must be written in a way which supports both MS-DOS and UNIX based operating systems, as both of these are used extensively in industry.

Design Requirements:

- The use of a programming language which is compatible with the API of the chosen cloud service provider. The language must also support object-orientation and be class based. Options include Java, C++ and Python.
- An Object-Orientation style design must be employed; Principles, such as a loose coupling of separate components, and code re usability need to be kept in mind when designing individual classes.
- The design of the system should be class based.

As the implementation of the Client/Server paradigm is required, it is necessary to gather requirements for each of the two components.

3.1.2. Client Component Requirements.

Functional Requirements:

The client will be installed on every new server which is instantiated with the aim of hosting the business application. It should essentially be a scanning system, which gathers data surrounding the performance of the server and the business application, and is then sent back to the server.

The client should be able to read the following metrics from the environment:

- Time of day – hourly, or as configured by the user.
- Application hits/second
- Exception counter: The client component should scan the server output log for any lines containing exceptions and severe warnings and increment a counter for each one.
- Exception type: Record the type of exceptions which are being output by the application regularly
- Number of Database query errors, or time-outs.
- Average CPU, Ram and Disk write speed,
- Bandwidth usage: record the Upload and Download speed of the network access interface in Megabytes per second (MB/sec)

The units measured by the client application will depend on the type of server on which it is hosted; For instance it would useless to measure database query errors on a web server

It is also required for certain parameters which have a role in the scanning process to be configurable, to allow for a flexible system:

- The Operating System in use – this will determine the location of the output log file.
- The threshold of the amount of times an exception or warning needs to be repeated before recording it
- The type of database installed – this will determine which query to run in order to gain information on the current state of the data source.
- The time scale for the scanning routine.

- Which metrics to scan during the routine.

Structural Requirements:

The client component should consist of two separate layers:

- The Service layer should run the scanning and sending routines regularly
- The Domain layer should provide the logic for fulfilling these routines.

Performance/Behavioural Requirements:

- The client application will need to be deployed to every new server which is created, this means that the system will need to be small and lightweight, with a quick start up time, to ensure it's maximum efficiency.
- The client component should be easily integrated with a custom server disk image.
- High scalability: as large amounts of data will be processed at given times, the application's performance should not diminish as its usage increases.

3.1.3. Server Component Requirements

Functional Requirements:

- A prediction algorithm will be at the centre of the system, and provide actions for any eventuality encountered in the environment. Specific requirements for this will be defined in a later section of this report.
- The sever component will provide a mechanism for converting the raw data received from the client into a format which is acceptable for the prediction algorithm, such as binary.
- The server will be given the functionality for taking action when presented with a specific input; the action set is: delete, add or modify a server, or do nothing
- The server must also take into account the instance type of new servers which are created, or servers which need to be modified.
- When creating a new server, the application must be able to deal with custom image files.

Similarly to the client application, the server must be given the functionality to allow users to change system variables, such as the prediction algorithm's variables.

Structural Requirements:

The Server should be created using four different layers, each of which has a clear and separate purpose:

- The Service layer contains the services which execute sequences of domain layer logic in order to carry out the desired functionality. This must include prediction algorithm training and cluster modification.
- The Domain layer contains all of the application logic used by the services.
- The mapper layer provides a link between the domain layer and the data source.
- The database contains the prediction data.

3.2. The Unified Modelling Language.

The Unified Modelling Language (Sinan Si, 1998) is a specification language for designing and documenting software applications. It is an I.T. industry standard for software design, and describes structural paradigms for the design process of object-oriented systems. The UML provides a variety of diagrams which are used to represent various aspects of a system, from different perspectives. For example, a UML class diagram is used to describe the structure of a system within the context of it's specific classes, and their corresponding attributes and operations, whereas the Use Case diagram should outline an application's flow from a user's perspective. It is also important to note that UML is implementation independent.

Use cases are to be employed when attempting to describe a system from a user's perspective. These diagrams outline the behavioural and procedural aspects of an application, as opposed to the structural aspects. Use case diagrams can be made of several elements:

- Actors: These are external elements which directly impact the system, and can either be a human user, or a separate system.
- Use cases: Represented by oval shapes, these describe a single behavioural requirement of the system.
- Use case relationships, such as the extends and communicates relationships
- The extends relationship describes a fundamental aspect of a system which is used repeatedly, and might be hidden by other use cases.

3.3 System Use Cases

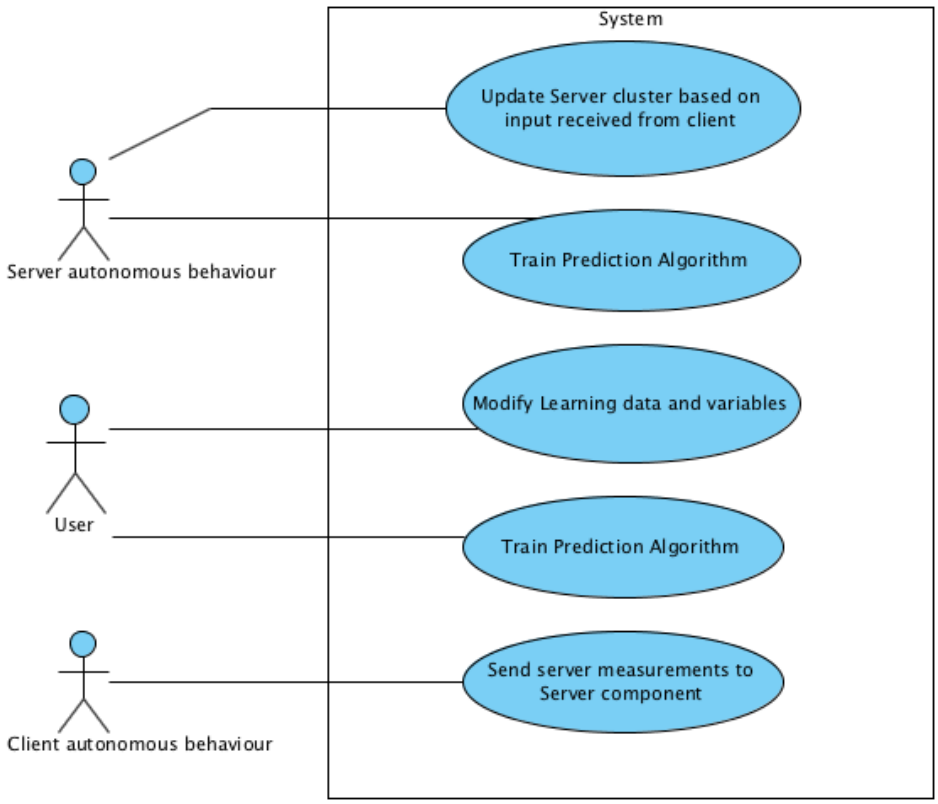


Fig. 9 UML Use Case Diagram of the Predictive Auto Scaling Software

Use Cases:

1. Through autonomous action, the server component updates the server cluster
2. The server component periodically trains the prediction algorithm.
3. The User modifies the prediction algorithm's learning data and variables
4. The User manually trains the prediction algorithm.
5. The Client Component sends measurements read from the server to the Server component of the application.

3.4 Predictive Learning Algorithm Requirements

As we are choosing to experiment with various Evolutionary Algorithms for the predictive element of the system, it is necessary to outline some functional requirements for the algorithms

Functional requirements:

- The Algorithms will adopt the Pittsburgh approach for evolving a rule base of condition : action rules.
- A suitable binary representation will need to be chosen for solutions.
- These representations will be encoded using Gray encoding; This will allow the algorithm to adhere to the basic principle of locality. Neighbouring genotypes should correspond to neighbouring phenotypes, which means that any modification to the genes of an individual should equate to a proportionate change in the individual's behaviour. The notion of generalisation must be handled, in order to classify the learning requirements successfully
- The Algorithm should be designed in a highly flexible way which allows users to modify it's learning parameters, allowing experimentation. This must include:
  - The ability to choose between the Darwinian, Lamarckian and Baldwinian evolutionary algorithms.
  - Variable crossover and mutation rates
  - Variable population size and tournament size.
- The population must be initialised randomly.
- Each individual will be created with a random set of genes; these can be decoded into specific input/action values, which define rules.
- A fitness function will evaluate the fitness of each individual, based on either a penalty or reward approach. Objectives and constraints must be user defined based on their specific scaling needs
- Selection, crossover and mutation operators must be implemented.
- A local search algorithm must be implemented, to be used by the memetic algorithms for their individual adaptation process.
- The algorithm should carry out these operations in an iterative fashion until an optimal solution has been found.
- Once the process is complete, the output of the algorithm should be a set of highly optimized server scaling rules.

Non functional requirement:

- The algorithm should be able to achieve the desired results within a sensible amount of iterations using a population of 100 individuals at most.

4. Software Design

4.1 Software Architecture

Following the requirements phase of the Waterfall development process, the design of the system can be described. The following diagram outlines the logical architecture of our system based on the previous architectural requirements.

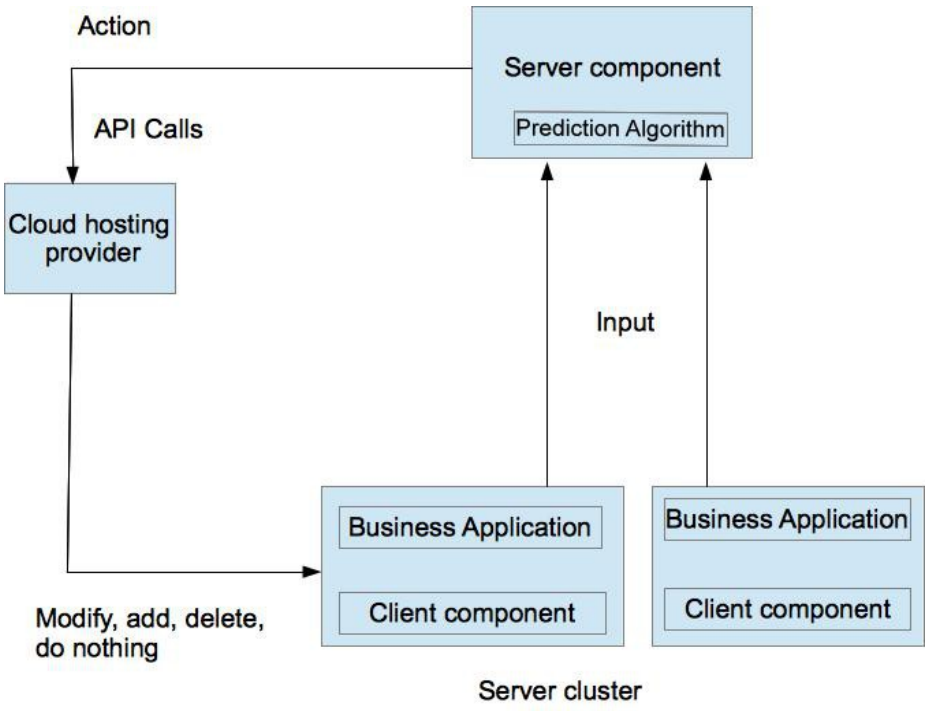


Fig. 10 Predictive Auto Scaling Software Architecture Logical Diagram

Software architecture logical diagram description:

1. The server component is unique, and contains the prediction algorithm
2. Client components run on each existing server in the cluster.
3. Clients read metrics from their environment, which are regularly sent to the server component.
4. Based on the input received, the server makes changes to the cluster as required.

Following the general structure of the system, we can proceed to individual component design.

4.2 Component Design

4.2.1 UML Class Diagrams

The Class Diagram is the most known and used UML paradigm. It describes the contents of the classes which form a system component, specifically their variables, constructors and methods. The class diagram also allows the visualization of the various possible interactions and associations between different classes. Class diagrams can be seen as the static description of a system, meaning they do not represent the behaviour of a system over time.

Within the Class Diagram model, a class must be described using a multiple tier representation, separated by lines:

- The class name, the first letter of which should be upper-case
- Attributes, which represent class variables. These must take the form of **variable: dataType** in the class diagram. The first letter of an attribute should be lower-case. Attribute visibility is represented as either a '+', '-' or '#' symbol, for public, private and protected visibilities respectively
- Constructors, represented as ConstructorName(parameterVariable: dataType)
- Operations, represented as OperationName(parameterVariable: dataType): returnDataType. The Operation visibility scheme is the same Attributes.
- Constructor and operations can have several parameters as required. Parameters are optional.

Some of the relationships between classes are defined as such:

- Generalisations: Describe links between a generalised class and its more specific implementations. These are represented as straight lines with a triangular arrow in the direction of the generalised class
- Dependencies: Describe a link between two classes, one of which requires data or knowledge from the other. Modifying the depended class results in a need for modifying the dependant. Dependencies are represented as broken lines with arrows in the direction of the depended class.
- Associations describe links between classes which contain common or similar functionality or behaviour. This implies that at least one of the two classes contains an object reference to the other. Associations are represented as continuous lines if both classes have references to each other. A directed association with a cross at its base implies that the association is only navigable in the direction of the arrow, meaning one class contains a reference to the other, but not vice-versa.

Using the principles of the UML Class Diagram, the structure of each component of the server scaling system can be defined. The classes are described here as they should appear in the system: Separated between client and server components, and structured as different tiers as required. The following sections provide a class by class description of each component.

4.2.2 Client Component Class Diagram.

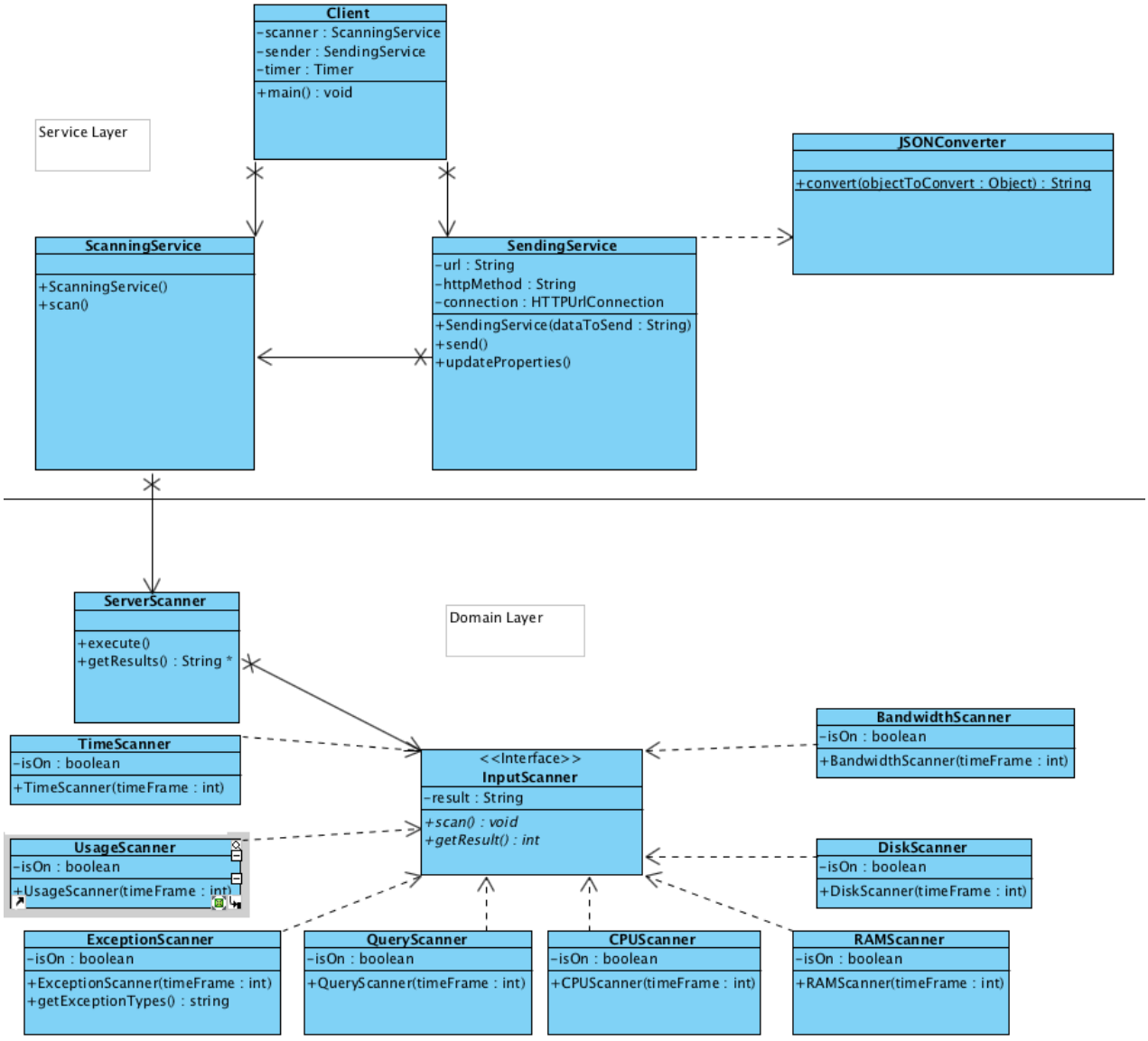


Fig. 11 UML Class Diagram of the Client Component's Service and Domain Layers

The Service Layer Classes:

Client: The main method of the application runs from this class, which instantiates and executes the two services which are included in this layer. The reference to Timer in this class is used to allow threads to be run repeatedly. As both services implement the Runnable interface, their code can be run as separate threads, on a regular basis, and in parallel to the rest of the application.

ScanningService: This class has as a reference to the ServerScanner which is contained in the domain layer, and allows the system variables which are used as criteria in the scaling process to be read.

SendingService: The class responsible for sending the data which has been scanned by the system to the Server Application. It is dependant on a properties file, which contains information such as the request URL and HTTP method type (POST) of the server component. The public method updateProperties() refreshes the class variables based on the properties file. The send() method contains all of the logic for running the request URL with the HTTP post parameters, which are a serialized string of the scanning results. This class also contains a Java.net.HTTPUrlConnection object, allowing URL's to be constructed and run. The sending service is also associated with the ScanningService, as the former requires the data to be sent to the server component.

JSONConverter: Allows a serializable object to be converted to the JSON format, using the static method convert(objectToConvert: Object). This is essential as JSON is one of the two formats (the other being XML) which are strictly only acceptable as HTTP POST parameters.

The Domain Layer Classes:

InputScanner: an Interface containing an abstract method scan(). Each class which implements this interface is responsible for providing the scanning process for a single specific system variable. The scan() method updates the result class variable as a String, which contains the value of the measured system variable once the scanning process is complete. The boolean isOn is found in every class which implements this interface, and is used to determine whether the class is currently active in the scanning process.

ServerScanner: This class contains references to each implementation of the InputScanner. The execute() method runs each InputScanner in a loop, and adds the result to a String array, thus allowing the collection of

results to be returned to the service layer in a single variable.

4.2.3 Server Component Class Diagram

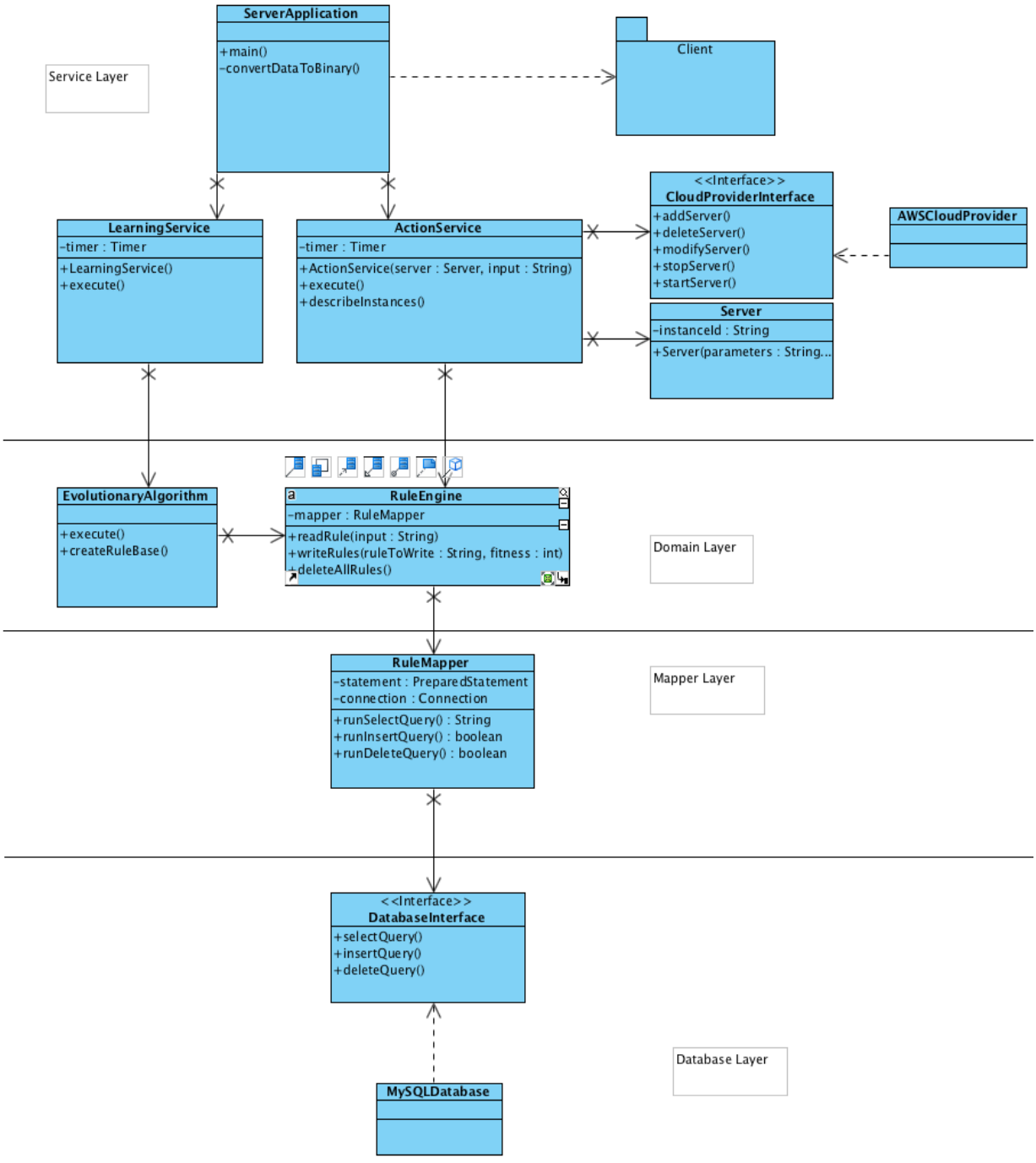


Fig. 12 UML Class Diagram of the Server Component's Service, Domain, Mapper and Database Layers

The Service Layer Classes:

ServerApplication: Contains references to both of the service classes which are contained in this layer of the system; LearningService.java and ActionService.java. This class also contains the main method of the application, from which both services are instantiated and executed. The convertDataToBinary() method allows the measurements posted by the Client component to the Server component to be converted to a binary string.

Server: Instances of this class are representations of Servers in the current cluster. They contain a reference to the instanceId of the server, in addition to other server parameters, which are contained in a string array.

LearningService: The service which allows for a scheduled execution of the Learning Algorithm which is contained in the Domain layer.

ActionService: This service allows actions to be carried out on the server cluster, depending on the input which has been sent from the Client Application. The execute() method calls the appropriate method on the implementation of the CloudServiceProvider interface, which allows the correct API call to be returned based on the input. The serialize() method allows the HTTP request to be formed. Finally, the request is sent to the host provider which updates the server cluster accordingly. This service does not run regularly in the same way the client application's services to, but simply reacts to the presence of new input which has been delivered by the client.

It is important that this class distinguishes when it is necessary to create and delete instances, or whether starting and stopping instances is sufficient. The software should never delete servers unless there is something fatally wrong with them, as doing so is less time and cost efficient. Prior to running a URL which performs an action, the ActionService will first need to run the DescribeInstances method, to check wether any stopped instances can be started again. From the prediction algorithm's level, it is assumed that the application will only be deleting instances and creating new ones. However if a stopped server is found to exist in the response of DescribeInstances, and the selected rule's action is to create a new server which is identical to the found server, the ActionService will simply wake the server rather than create a new one.

CloudProviderInterface: A generic interface which allows the software to remain independent of any third party virtual server hosting provider. Implementations of this class need to provide the details of the API calls which are used by the ActionService for taking scaling actions. This interface is necessary as different

cloud providers will require different API calls for the same action.

AWSCLoudProvider: An example of a potential implementation of this interface is provided here, based on the Amazon Web Services API.

HTTP requests to the Amazon Web Services API are built in the following format:

```
http://ec2.amazonaws.com/?Action=StartInstances&InstanceId.1=i-10a64379&AUTHPARAMS
```

(Amazon, 2014)

The Action HTTP parameter refers to the scaling action to take, and is followed by any other required parameters.

The following table describes all actions which will need to be handled by the methods implemented in the ASWCloudProvider class

Interface Method	AWS HTTP Action	Required HTTP Parameters	Effect
addServer()	RunInstances	• ImageId: Specifies which AMI (Amazon Machine Image) is to be used. This refers to the operating system image which needs to be built and run upon server startup • AuthParams: The authentication parameters for the service, username and password	Creates a new Server Instance
startServer()	StartInstances	• InstanceId: The instance number of the previously created instance. • AuthParams	Starts a Server Instance
stopServer()	StopInstances	• InstanceId • AuthParams	Shuts down an instance but does not terminate it
deleteServer()	TerminateInstances	• InstanceId • AuthParams	Terminates an instance. This can no longer be rebooted
modifyServer()	StartInstances	• InstanceId • InstanceType: Allows the user to • specify a new instance type. • AuthParams	Modifies a previously stopped server

The Domain Layer Classes:

PredictionAlgorithm: This class provides the functionality for learning rules. It contains a reference to the RuleEngine class, which allows new highly fit rules which have been discovered through the learning process to be added to the database. The PredictionAlgorithm class should cover any possible environment input, and provide a corresponding action.

RuleEngine: Provides the functionality for reading, adding and deleting rules from the database. It contains a reference to the RuleMapper.java class, which is essentially a link between the Domain layer and the database.

The Mapper Layer Class:

RuleMapper: This mapper class contains dynamic database queries which are constructed depending on the values which are passed to it from RuleEngine.java, and the query structures contained in the DatabaseInterface implementation. This is made possible by using the PreparedStatement class, which allows SQL queries to be built, and then executed against a database, using a java.sql.Connection object. The main advantage of using the PreparedStatement object is that it reduces execution time when running a large amount of queries, as only a single object is needed to build, compile and run a single query. Similarly, only a single JDBC Connection object is needed throughout the entire process; This class can be found in java.sql, and requires three parameters to be passed in: The user name and password needed to connect to the database (the user must have read and write permissions) and the database's URL.

The Database Layer Class:

DatabaseInterface: A wrapper class which allows the application to remain data source independent. Implementations of this interface need to provide the specific query syntax for querying any database a user might want to employ. Two example implementations are given here, using MySQL and SQL server. However these are not exhaustive, as this interface handles any other possible data sources.

4.3 Evolutionary Algorithm Design

The following aims to describe a possible design for the prediction algorithm of the system, using Evolutionary Algorithms to evolve rule bases of if:then rules. The generated rules should classify server measurements based on suitable actions, which should be taken when such measurements are read by the system. This component is designed in such a way which allows for the Darwinian, Lamarckian and Baldwinian to be implemented, in order to eventually compare their learning results. Solution representation design will be discussed first, followed by class structure design of the three algorithms.

4.3.1 Solution Representation

Individual genes are made of input:action rules. The input corresponds to server measurements, and the action corresponds to the required modification to be made to the server cluster. Rules should be independent of servers, meaning they do not contain information such as a server ID. Each rule should be applicable to any server of the same type. However, certain applications might require different scaling criteria for different types of servers, such as application or database servers. In this case, learning criteria should be defined for each server type, and different rule bases should be generated. This representation can be used by the three Evolutionary Algorithms.

In this case, binary strings would be the most suitable choice for solution representation. Using a real valued representation would have significant drawbacks: As the software is tracking various server resources which are not measured in the same manner, an individual's representation based on real values would not be consistent. It is therefore more efficient to convert the measured value to a binary string, which will allow the genetic operators to be applied with ease.

Based on the scaling metrics previously defined, Each rule can consist of the following measurements

Measurement	Number of bits	Maximum Decimal Value	Constrained value range
Hour of Day	5	31	25 -31
Exception Counter	7	127	100-127
Dominant Exception Type	4	NA	NA
CPU %	4	15	10-15
Ram %	4	15	10-15
Disk Write Speed %	4	15	10-15
Database Errors	7	127	100-127
Application hits	7	127	100 - 127
Bandwidth Usage	7	127	100-127
TOTAL	**49**		

The constrained value range column refers to the range of feasible values for each measurement, given our representation.
For example, for the hour of day measurement, the range 25 – 31 is an impossible value. As we can see from this table, the representation allows for a number of infeasible values, meaning that a suitable constraint handling mechanism will be required.

The dominant exception type measurement has a direct mapping scheme, depending on the language of the business application which is hosted on the cloud infrastructure. This would translate common exception types of the language which the business application is written in. For example a possible exception mapping for Java could be:

1001 : java.lang.NullPointerException

As all of these measurements are optional. It is important that all members of the algorithm's population have an equal length. This means that for different server types requiring a different rule base for prediction, a new instance of the prediction algorithm will need to be generated.

The final 2 action bits are required, bringing the maximum possible length of a rule to 51. The action bits can be predefined as such

10: Add server
01: Delete server
11: Modify server
00: No action

The final action value is not a constrained value, it simply means that given an input which matches this action, server scaling is not required

An example of a rule might be

1010000 1100 : **01**

Which represents:

- Any hour of the day
- 5 application hits
- 30% CPU usage
- Action: Delete server

Spaces and colons are used here for readability purposes, these should not appear in the actual representation.

We have opted for the Pittsburgh approach, as the maximum length which an individual can take using this method is 510, provided the rule base which is being generated contains 10 rules, meaning computation will not be too expensive.

4.3.2 The Fitness Function and Constraint Handling

For the task at hand, we cannot define a significant amount of constraints for the fitness function to have the ability to make subtle differences in solution quality based on these constraints alone. However, we can describe scaling objectives, as these will represent a user's scaling requirements which will be very specific to the user's needs. A combination of both of these methods should be employed, but reward system should

have a much greater influence on the fitness of an individual, as the number of objectives are fewer, yet more specific than the constraints. We still wish to significantly penalise any individual which contains invalid rules, to the point where it is unlikely to be reselected for future generations. Using this method we can handle constraints from within the fitness function.

The evaluation process should reward individuals which successfully classify the user defined learning data; If an individual contains a rule made of a generalised input string which matches a data entry, their action strings are compared. If the latter are also equal, the individual in question is awarded a fitness increase. For example:

A data entry which corresponds to a scaling requirement could be as such:

1000100111110 : 01

And an individual contains a rule which is represented as:

100#100###110 : 01

This would result in the individual containing the previous rule being rewarded.

For a solution to be optimal, it must contain a rule which classifies the entire data set as such. If we assume that an individual's fitness is set to 0, and is rewarded with a single fitness incrementation for each successful classification, the fitness of an optimal solution must always be exactly equal to the number of data entries in the scaling requirements.

4.3.3 Evolutionary Operators.

The tournament selection and uniform crossover methods are used in this design. Furthermore, the Hill Climbing local search method is used by the two memetic algorithms, as we are looking to only slightly improve individuals through adaptation and the search for optimal solutions can be handled by the evolutionary process

Following solution representation design, we can outline the class structure of the three algorithms

4.3.4 Evolutionary Algorithm UML Class Diagram

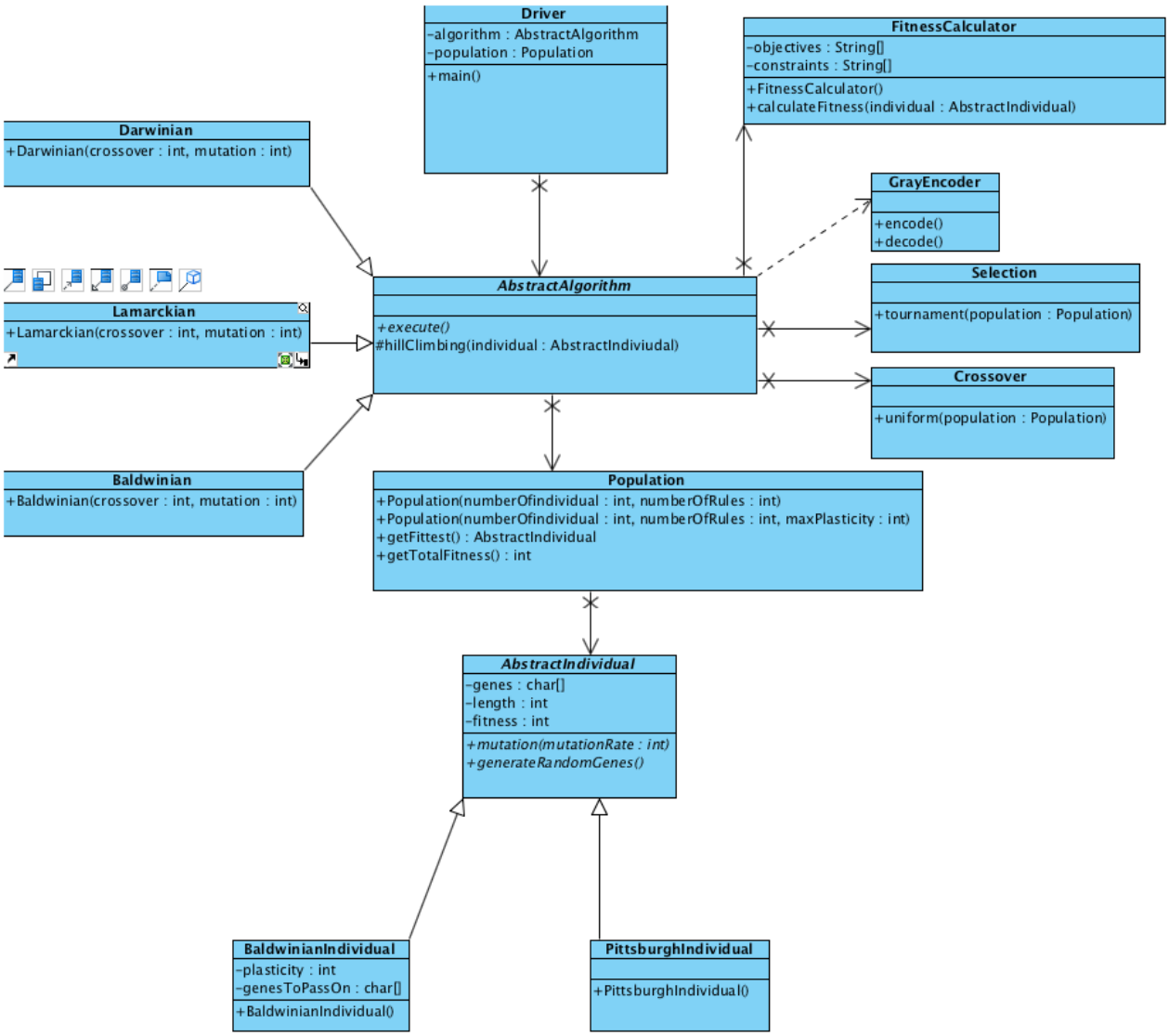


Fig. 13 UML Class Diagram of the Darwinian, Lamarckian and Baldwinian Evolutionary Algorithms

AbstractAlgorithm: An abstract class which is extended by all three variations of the genetic algorithm. Contains the abstract method execute which must be implemented by its sub-classes. The hillClimbing() protected method is called in the Baldwinian and Lamarckian algorithms. This class has no constructor and can not be instantiated.

Darwinian, Lamarckian and Baldwinian: These classes all extend AbstractAlgorithm.java and implement the execute() method in different ways, depending on the requirements for the specific algorithm. The constructors for these algorithms take two integers as parameters; the mutation rate and crossover rate.

GrayEncoder: A static class which contains the algorithms for encoding to and decoding from Gray Encoding.

Selection: Contains the tournament selection method which requires a Population object to be passed as a parameter

Crossover: Contains the uniform crossover method which requires a Population object to be passed as a parameter

FitnessCalculator: Instantiated to calculate the fitnesses of individuals. Contains class variable references to the constraints and objectives which are used to determine the fitness.

AbstractIndividual: An abstract class containing the genes of an individual which are represented as a char array. Also contains references to the length of an individual, which is the number of indexes in the genes char array, and the individual's fitness. The mutation(mutationRate: int) and generateRandomGenes() methods are abstract and must be implemented by all sub-classes.

PittsburghIndividual and BaldwinianIndividual: These classes extend AbstractIndividual.java. They have been separated in this design as their structures are different, meaning that the mutation and generateRandomGenes methods must be implemented in different ways. The PittsburghIndividual class is used by both the DarwinianAlgorithm and LamarckianAlgorithm classes, whereas BaldwinianIndividual is solely referenced in BaldwinianAlgorithm. The plasticity class variable is only used within the BaldwinianAlgorithm class. The BaldwinianIndividual class also contains a String variable named genesToPassOn. This is used to reference individuals' non adapted genes which are carried on to future

generations.

Population: This class extends an ArrayList of AbstractIndividual. It contains three separate constructors for each Individual type. Population(numberOfindividual : int, numberOfRules : int) is used in the Darwinian and Lamarckian algorithms, as it instantiates a populaiton of PittsburghIndividuals.
Population(numberOfindividual : int, numberOfRules : int, maxPlasticity : int) is only used within the BaldwinianAlgorithm class, as it randomly assigns a plasticity to each BaldwinianIndividual it instantiates.

4.4 Deployment

As the Server component's database is not large, it does not require a separate machine to the server application. These should be deployed in conjunction to a single machine.
A custom server image should be created by the user which contains pre installed versions of both the Client component and the business application. This should be submitted to the chosen third party hosting service, and the image should be referenced when the system's server component creates a new server within the cluster.

5. Software Implementation

The following describes a possible implementation for the server component's domain, mapper and database layers, using the Java platform and a MySQL relational database. The Darwinian, Lamarckian and Baldwinian algorithms are implemented, in order to compare their performance, and ultimately decide which is the most efficient at solving the task at hand.

5.1 Implementation Tools

Java:

The Java programming language has been employed, as it offers many advantages over others:

- Java is a strongly typed programming language which is compiled as opposed to interpreted, meaning it has a very strict syntax. This allows run time syntax errors to be avoided.
- Java is an open source platform, meaning a developer license is not required.
- It is platform independent, meaning it can run on any operating system providing the Java software platform is installed.
- Java is class based, making it ideal for Object-Oriented software.

Netbeans IDE:

The Netbeans Integrated Development Environment provides the tools for editing, compiling and debugging software. It supports many languages and platforms such as Java, HTML and PHP, amongst many others. By combining these utilities into a single piece of software, Netbeans improves efficiency throughout the development process. It's Graphical User Interface increases user productivity, as the use of a command line when compiling programs can be avoided completely.

MySQL and MySQL Workbench:

MySQL is a Relational Database Management System (RDMS) which is based on the Structured Query Language (SQL) syntax for creating and editing relational databases. MySQL is open source, and is easily integrated with Java through using a single library. MySQL Workbench provides a GUI for editing and building MySQL databases, meaning the use of a command line can be avoided.

5.2 Implementation Extracts

Following the UML class design outlined in the Design section, the implementation of many of the core classes contained in the server's domain and database layers are described here, using screen captures of code extracts and line by line explanations of the code.

5.2.1 Implementation of the Evolutionary Algorithms

The Following shows how the Darwinian, Baldwinian and Lamarckian algorithms can be implemented by extending the AbstractAlgorithm class defined in the design phase. Although the Lamarckian and Baldwinian memetic algorithms use the concept of memes as their representation, they are still referred to as genes in this implementation, for code re usability reasons.

The Darwinian Algorithm.

```java
@Override
public void execute()
{   // Loop for specified generations
    for (int index = 0; index < numberOfGenerations; index++)
    {
        // calculate each individual's fitness
        for (AbstractIndividual individual: population)
            fitness.calculateFitness(individual);

        while (currentGeneration.size() < population.size())
        {   //Select population to reproduce using tournament selection
            AbstractIndividual individualToAdd = selection.tournamentSelection(population, tournamentSize);
            currentGeneration.add(individualToAdd);
        }
        instantiateIndividualToKeep();  //Elitism

        for (AbstractIndividual individualToEncode : currentGeneration)
        {   //Gray encode individuals before reproduction
            char[] encoded = GrayEncoder.encode(individualToEncode.getGenes());
            individualToEncode.setGenes(encoded);
        }

        //Perform the crossover
        crossover.uniform(currentGeneration);

        for (AbstractIndividual individualToEncode : currentGeneration)
        {   //Individual mutation
            individualToEncode.mutation(mutationRate);
            individualToEncode.actionMutation(actionMutationRate);
            //Decode each individual
            char[] decoded = GrayEncoder.decode(individualToEncode.getGenes());
            individualToEncode.setGenes(decoded);
        }

        double averageFitness = currentGeneration.getTotalFitness() / (double) currentGeneration.getPopulationSize();
        //Print the average fitness and best fitness of the generation
        System.out.println(averageFitness + "  " + population.getFitest().getFitness());

        if(index == numberOfGenerations - 1)
            System.out.println(currentGeneration.getFitest().toString());

        population.clear();
        //The new generation replaces the old one
        for (AbstractIndividual individual: currentGeneration)
            population.add(individual);

        population.remove(population.getLeastFit());
        population.add(individualToKeep);
        currentGeneration.clear();
    }
}
```

Fig 14. Code Extract of DarwinianAlgorithm. java

Each of the separate evolutionary algorithms extend the AbstractAlgorithm.java class, meaning they must override and implement it's abstract method execute(). The previous code extract is of the execute() method implemented in DarwinianAlgorithm.java. The class variables population, currentGeneration, selection, crossover, fitness and configuration are instantiated in the super-class, meaning they do not need to be defined in each subclass. This makes use of the concept of code re usability, which is essential in any object-

oriented applications requiring simple maintenance.

This method implementation contains the procedural logic of the Darwinian algorithm:

- 4: An initial for loop repeats the process for the amount of generations required.
- 7-8: A nested for loop calculates the fitness of each member of the population in turn.
- 10-14: The following nested while loop selects individuals from the global population, and adds them to the current generation until the population size requirement, which is defined in the constructor, has been met
- 15: The fittest individual of the population is then selected, and instantiated as a new Individual object. This functionality is encapsulated in the instantiateIndividualToKeep() method, of return type void, which is contained in the super-class. A new object reference must be instantiated for the individual to keep, as it must not have the genetic operators applied to it.
- 17-21: Prior to applying the crossover and mutation operators, a new for loop selects each member of the population and encodes their genes, by calling the static method encode on GrayEncoder.java
- 24: The uniform crossover method is called, with the current population passed in as a parameter.
- 26-33: A third for loop contains a call to the mutation method, which is applied to each Individual object in turn. In the same loop, each individual's genes are decoded from Gray encoding.
- 35-37: The average fitness and best fitness of the generation are output to the console to inform the user
- 42-49: Finally the population object is cleared, and the contents of the currentGeneration object are copied. The best current individual is also added to the population object for the purpose of elitism.

The initialisation of the algorithm's class variables in AbstractAlgorithm can be seen here

```
public AbstractAlgorithm (int numberOfGenerations, int crossoverRate, double mutationRate,
        double actionMutationRate, int populationSize, int tournamentSize, String individualType)
{
    this.numberOfGenerations = numberOfGenerations; // Number of iterations of the algorithm
    this.mutationRate = mutationRate; //Individual mutation rate
    this.tournamentSize = tournamentSize; // Tournament size for selection
    this.actionMutationRate = actionMutationRate; //Mutation rate for rule actions
    this.individualType = individualType; // Either simple pittsburgh all Baldwinian pittsburgh
    this.populationSize = populationSize; // The total size of the population

    selection = new SelectionMethod(); // Used for performing selection
    crossover = new CrossoverMethod(crossoverRate); //Used for performing crossover
    fitness = new FitnessCalculator(configuration); //Used for calculating individual fitness
    configuration = new Configuration(); // Contains the learning data

    if(this instanceof DarwinianAlgorithm || this instanceof LamarckianAlgorithm)
        population = new Population(populationSize, numberOfRules);
    //Separate constructor for Baldwinian individuals with the max plasticity rate
    else if(this instanceof BaldwinianAlgorithm)
        population = new Population(populationSize, numberOfRules, 100);

}
```

Fig 15. Code Extract of AbstractAlgorithm. java

The class variables are initialised in this constructor, meaning separate constructors do not need to be defined for each concrete class. Two separate constructors for the Population class can be called within it, depending on the type of Algorithm being used. As the Baldwinian algorithm takes into account the notion of plasticity, it requires a separate constructor for it's population which defines the maximum plasticity for an Individual, in this case, 100.

The Lamarckian Algorithm

The Lamarckian and Baldwinian algorithms implement the execute method with almost exactly the same logic, with some differences for handling the local search mechanism. In this implementation, the localSearchRate variable defines the probability of running a hill climber local search algorithm on an individual's genes, in order to find the fittest solution in it's surrounding neighbours. This class variable is instantiated to 90. Using a rate of 100% is not desirable, as it results in the population loosing it's diversity.

```
1   @Override
2   public void execute()
3   {
4       for (int index = 0; index < numberOfGenerations; index++)
5       {
6           for (int individualIndex = 0; individualIndex < population.size(); individualIndex++)
7           {       //Perform local search on 90% of the population
8                   if(random.nextInt(100) < localSearchRate && !population.get(individualIndex).equals(population.getFitest()))
9                       localSearch(population.get(individualIndex));
10                  //If individual doesn't adapt calculate fitness normally
11                  else
12                      fitness.calculateFitness(population.get(individualIndex));
13          }
```

Fig 16. Code Extract of LamarckianAlgorithm. java

- 6: Each individual of the population is looped over
- 8 - 9: The random variable, an instance of Random.java, produces a value which is compared to the local search rate. If this value is within the rate, the local search is executed
- 11 – 12: In the case in which the produced value is not within the local search rate, the individual's fitness is calculated normally.
- Individuals keep the genes which result of running the local search algorithm.
- The rest of the algorithm remains the same as the Darwinian algorithm.

The Baldwinian Algorithm:

An individual based on the Baldwinian approach has the possibility to adapt to its environment, depending on the value of it's plasticity. It is essential that an individual's adapted genes are not passed on to future generations. Each individual is assigned a random plasticity rate when initialised, which is subject to change throughout the evolutionary process through mutation. This is translated in the implementation of the evaluation process shown here:

```
1  @Override
2  public void execute()
3  {
4      for (int index = 0; index < numberOfGenerations; index++)
5      {
6          for (AbstractIndividual individual: population)
7          {
8              BaldwinianIndividual individualToUse = (BaldwinianIndividual) individual;
9
10             char[] genes = individualToUse.getGenes();
11             //Individual adaption based on plasticity rate
12             if(random.nextInt(100) <= individualToUse.getPlasticity() && !individualToUse.equals(population.getFitest()))
13                 localSearch(individualToUse);
14             //If the individual doesn't adapt calculate fitness normally
15             else
16                 fitness.calculateFitness(individualToUse);
17             //Adapted genes are not passed on to the next generation
18             individualToUse.setGenesToPassOn(genes);
19         }
```

Fig. 16 Code Extract of BaldwinianAlgorithm. java

- 4: Each individual of the population is looped over
- 8 – 10: The genes of each member of the population are stored in a separate char array variable prior to calculating the fitness
- 12 – 16: The random variable used here produces a value which is compared to the individual's plasticity variable. If the probability is met, the local search algorithm is called. Otherwise, the individual's fitness is calculated as it is.
- 18: The genes char array variable is then assigned to the individual's genesToPassOn variable, which is used during the crossover and mutation phases. This means that any genes which are the result of the local search process are not passed on to children.

Rather than describing the implementation class by class, the functionality used in the execute() method will each be described in turn, in order to outline the flow of the algorithm

5.2.2 Implementation of the Algorithm Operators

These include the selection, crossover, mutation and hill climbing methods and are common to all algorithm variations.

The return type for this method is AbstractIndividual, representing the individual which has been selected. The population is passed to the method as a parameter, in addition to the tournament size.

```
1   public AbstractIndividual tournamentSelection(Population population, int tournamentSize)
2   {   //Create the tournament
3       Population tempPopulation = new Population();
4
5       for(int index = 0; index < tournamentSize; index++)
6       {   //Randomly add individuals to the tournament until the tournament size has been reached
7           int individualIndex = generator.nextInt(population.getPopulationSize());
8           tempPopulation.add(population.get(individualIndex));
9       }
10      //The fittest individual of the tournament is selected
11      return tempPopulation.getFitest();
12  }
```

Fig 17. Code Extract of Selection. Java

- 3: A temporary Population object is constructed which will contain the tournament.
- 5: A for loop is run until the tournament size has been reached
- 7 – 8: Individuals are selected at random; the generator local variable here represents an instantiation of the Java Random class. These are added to the tournament, the tempPopulation object.
- 11: The fittest individual of the tempPopulation object is returned.

The Uniform Crossover Operator:

This method is contained within the Crossover class, and requires a Population object to be passed as a parameter, which represent the individuals to which the crossover operation will be applied.

```
1   public void uniform(Population individuals)
2   {
3       for(int index = 0; index < individuals.size(); index = index + 2)
4       {   //Loop over population in pairs
5           AbstractIndividual individualOne = individuals.get(index);
6           AbstractIndividual individualTwo = individuals.get(index + 1);
7
8           //Loop over each gene index of the pair
9           for(int i = 0; i < individualOne.getGenes().length; i++)
10          {
11              int random = generator.nextInt(100);
12              //Each gene has a chance of crossing over
13              if(random <= crossoverRate)
14              {
15                  //Perform the crossover
16                  char tempGene = individualOne.getGenes()[i];
17                  individualOne.getGenes()[i] = individualTwo.getGenes()[i];
18                  individualTwo.getGenes()[i] = tempGene;
19              }
20          }
21      }
22  }
```

Fig. 18 Code Extract of Crossover. java

- 3 - 6: A for loop selects pairs of individuals at a time, these are stored in two local variables and represent two parents, which will form new solutions.
- 9 – 11: A nested for loop selects single genes at the same index of the pair. Each gene of an individual has a chance of crossing over with it's counterpart. The generator class variable represents an instance of the java.util.Random class.
- 13 – 19: An if statement evaluates whether the next randomly generated integer is within the probability rate. If this is the case, the crossover is performed. Otherwise, the same process is repeated for the next pair of genes.

The Mutation Operator:

This method is contained within the AbstractIndividual.java class. The mutation rate is passed to the method as a parameter and the method's return type is void.

```
1   @Override
2   public void mutation(double mutationRate)
3   {
4       int count = 0;
5
6       for(int index = 0; index < getGenes().length; index ++)
7       {
8           count ++;
9           //Algorithm does not mutate action bits
10          if(count == getRuleLength() - 1)
11              continue;
12          //Reset the counter if second action bit is reached
13          if(count == getRuleLength())
14          {
15              count = 0;
16              continue;
17          }
18          //Each gene has a chance of mutating
19          if(randomGenerator.nextDouble() > mutationRate)
20              continue;
21
22          int rand = randomGenerator.nextInt(2);
23          //Perform the mutation based on rand variable
24          switch(genes[index])
25          {
26              case '0':
27                  if(rand == 0)
28                      genes[index] = '1';
29                  else
30                      genes[index] = '#';
31                  break;
32
33              case '1':
34                  if(rand == 0)
35                      genes[index] = '0';
36                  else
37                      genes[index] = '#';
38                  break;
39
40              case '#':
41                  if(rand == 0)
42                      genes[index] = '0';
43                  else
44                      genes[index] = '1';
45                  break;
46          }
47      }
```

Fig. 19 Code Extract of AbstractIndividual.java

- 6: Each gene of the individual is looped over.
- 10 – 17: The count integer represents the current index of the individual's input bits. This variable is necessary in order to avoid mutating any of the gene's action bits. This is handled in a separate

method which does not allow genes to be mutated to invalid actions. The loop simply ignores individual action bits, and proceeds to the next genes.

- 19 – 20: The class variable randomGenerator produces a random double, which is compared to the mutation rate. If the probability is not met, the algorithm continues to the next gene index.
- 22: In the case in which the mutation probability is met, a binary random integer is generated, which represents the value of the mutation to occur.
- 24: The genes class variable is a char array representing the genes of the individual in question
- 24 – 46: A switch statement executes the mutation, by selecting the gene value at the specified index, and assigning a new value to the gene based on the value of the randomly generated binary integer
- This process is repeated for every index of the char array.

The Hill Climber Local Search Algorithm

This method has a return type of void, and requires an Individual object to be passed as a parameter

```
1  private void hillClimber(PittsburghIndividual individual)
2  {
3      double originalFitness = individual.getFitness();
4      final char[] originalGenes = individual.getGenes();
5      int count = 0;
6      int ruleLength = ((PittsburghIndividual)individual).getRuleLength();
7      //Loop over each gene of the individual
8      for(int index = 0; index < individual.getGenes().length; index++)
9      {
10         count ++;
11         //Action bits should not be modified
12         if(count == ruleLength - 1)
13             continue;
14
15         if(count == ruleLength)
16         {
17             count = 0;
18             continue;
19         }
20         //Modify the gene to evaluate neighbour
21         char[] modifiedGenes = flipSingleBit(individual.getGenes(), index);
22
23         individual.setGenes(modifiedGenes);
24         //Calculate neighbour's fitness
25         fitness.calculateFitness(individual);
26
27         if(individual.getFitness() > originalFitness)
28         {
29            individual.setGenes(modifiedGenes); //Neighbour replaces current solution
30            break;
31         }
32         else
33         {   //Neighbour is not an improvement
34             individual.setGenes(originalGenes); //Individual doesn't change, evaluate next neighbour
35             individual.setFitness(originalFitness);
36         }
37     }
38 }
```

Fig. 19 Code Extract of AbstractIndiidual.java

- 3 – 4: The current fitness, and the genes of the individual are saved to local variables
- 8 – 19: A for statement loops over each gene of the individual. Similarly to the mutation operator, modifying action bits must be avoided. These are tracked by the count integer.

- 21: The gene at the current index is randomly modified, in a similar fashion to mutation
- 25: The individual is re-evaluated with it's newly modified genes.
- 27 – 31: An if statement evaluates whether the individual's fitness has improved as a result of modifying it's genes. If this is the case, the individual remains as it is, with the modified genes, and the process is terminated
- 33 – 36: Otherwise, the individual is reinitialised to its original state, and the process is repeated for the next gene index.

The Fitness Function:

This method requires a single parameter, which represents the individual being evaluated. The return type is void

```
1   private void calculatePittsburghFitness(PittsburghIndividual individual)
2   {
3       int fitness = 0;
4       //Load the learning data, loop over each data entry
5       for(String objective: configuration.getObjectives())
6       {
7           char[] objectiveBits = objective.toCharArray();
8           //Loop over each rule of the individual
9           for(int geneIndex = 0; geneIndex < individual.getGenes().length; geneIndex = geneIndex + individual.getRuleLength())
10          {
11              int length = individual.getRuleLength();
12              char[] individualGenes = new char[length];
13              System.arraycopy(individual.getGenes(), geneIndex, individualGenes, 0, individual.getRuleLength());
14
15              // If the rule doesnt match the data entry loop to next rule
16              if(!compareIndividuals(0, objectiveBits.length - 2, objectiveBits, individualGenes))
17                  continue;
18
19              //Penalty for rules with invalid action
20              if(individualGenes[length - 2] == '1' && individualGenes[length - 1] == '1')
21                  continue;
22
23               // If the action bits match the data entry has been successfuly classified
24              if(individualGenes[length - 2] == objectiveBits[length - 2] && individualGenes[length - 1] == objectiveBits[length - 1])
25              {
26                  fitness++; //Individual is rewarded
27                  break;
28              }
29          }
30      }
31
32      individual.setFitness(fitness);
33  }
```

Fig. 20 Code Extract of FitnessCalculator.java

- 3: The fitness variable is initialised to 0

- 5: The server scaling objective learning data is loaded from the configuration file, and each data entry is looped over
- 9: A nested for loop selects portions of the individual which represent the evolving rules. The loop's index is incremented by the length of a rule for each iteration
- 11 – 13: The individual's genes at the specified indexes are copied into a temporary char array, individualGenes.
- 16 – 17: This if statement compares the input bits of an individual to the current data entry. In a case in which these do not match, the algorithm continues to the following rule portion of the individual. Otherwise, the action bits of the data entry and the potential rule are compared
- 20 - 21: The second if statement evaluates whether the current rule's action bits are equal, to '11'. In this case, the rule is discarded, as '11' is not a valid action, and the loop continues to the next rule.
- 24 – 28: Finally, if the rule's action bits are equal to the data entry's action bits, the individual is awarded a fitness incrementation, as it has successfully classified the data entry. The algorithm then proceeds to eluate the following data entry, and not the next rule portion. This is essential, as it allows the algorithm to discard repeated rules within the individual. If the data entry's action bits have not been matched by the individual, the algorithm proceeds to the next rule portion, and not the next data entry.
- 32: Once the selection of data entries has been exhausted, the calculated fitness is assigned to the fitness class variable, in order to be retrieved later on in the process.

5.2.3 Implementation of the Database Layer

The database contains a single table, rule, which resembles the following when viewed in MySQL workbench:

id	input	action
49	#10###0###0#1###	10
50	01#0#0#010101##1	10
51	001#1#0###11#0#0	10
52	#####1#1#00#1###	01
53	101#110#00#010#0	10
54	#010##110##010#0	00
NULL	NULL	NULL

Fig. 21 Extract of the MySQL rule table

The table contains two columns of SQL VARCHAR data types, one containing the generalised inputs of the rules, and the other containing the corresponding actions.

5.2.4 Implementation of the Mapper Layer

The RuleMapper.java class provides the implementation for mapping rule data between the data source and the domain objects, and is contained within the mapper layer of the application. It's methods include:

- selectQuery(): Allows the rules in the database to be returned to the domain layer
- deleteQuery(): Deletes the current rule base
- insertQuery(String ruleToAdd): Writes a rule from the domain layer to the database
- countQuery(): Returns the number of rules in the database to the domain layer.

Rule Selection:

This method returns a HashMap containing all rules currently in the database. The HashMap keys refer to rule conditions, and the corresponding HashMap values refer to the rule actions.

```
1   public HashMap<String, String> selectQuery()
2   {
3       HashMap rules = new HashMap();
4       ResultSet results = null;
5       //The query to run
6       String query = "SELECT input, action FROM rules ";
7
8       try
9       {   //Prepare and execute the query
10          PreparedStatement statement = connection.prepareStatement(query);
11          results = statement.executeQuery();
12
13          while(results.next())
14          {   // Add each selected if:then rule to a HashMap
15              rules.put(results.getString("input"), results.getString("action"));
16          }
17
18      }
19      catch (SQLException ex)
20      {
21          Logger.getLogger(RuleMapper.class.getName()).log(Level.SEVERE, null, ex);
22      }
23
24      return rules;
25  }
```

Fig. 22 Code Extract of RuleMapper.java

- 10 : The class variable connection refers to a connection object obtained through the java.sql.DriverManager class. The latter requires the URL of the database in addition to login credentials to connect to the database. In order to import classes from the java.sql library, a Java MySQL connector library must be included in the project class path. Without this, connecting to the

database will not be possible. The java.sql.PreparedStatement object readies the query to be run on the database

- 11 : The query is then run by calling the mapper's executeQuery() method, which returns a java.sql.ResultSet object.
- 13 – 16: The result set is looped over to select each data entry. Any data contained in the 'input' column of the rule table is used as HashMap key, and data in the 'action' column is used as the corresponding values. These are passed to the HashMap's put method as parameters
- 19 – 22: Any SQL exceptions, such as syntax or connectivity errors must be handled.

Rule Insertion:

This method has a return type of void, and requires one String parameter, which represents the rule to be added to the data source.

```
1  public void insertQuery(String ruleToAdd)
2  {
3      //Template of query to run
4      String query = "INSERT INTO rules (input, action) VALUES ( ? , ? )";
5
6      String input = ruleToAdd.substring(0, 16); // Rule if portion
7      String action = ruleToAdd.substring(16).replaceAll(":", ""); //Rule then portion
8      action = action.replaceAll(" ", ""); //Sanitize the string
9
10     try
11     {  //Prepare the query
12         PreparedStatement statement = connection.prepareStatement(query);
13         statement.setString(1, input); //Replace first question mark with rule input
14         statement.setString(2, action); //Replace second question mark with rule action
15         statement.executeUpdate(); //execute the query
16     }
17     catch (SQLException ex)
18     {
19         Logger.getLogger(RuleMapper.class.getName()).log(Level.SEVERE, null, ex);
20     }
21
22 }
```

Fig. 23 Code Extract of RuleMapper.java

- 4: The query to be run is initially built, with question marks representing the data values to be inserted. This allows the query to be built dynamically, through the PreparedStatement object, based on the parameter which is passed in to the method
- 6 – 7: The string is separated into it's corresponding input and action using the substring method.
- 8: The resulting strings are sanitized, in order to remove any prior formatting made in the domain layer.
- 12 – 14: A PreparedStatement object is instantiated, and using the class' setString method, the query can be built dynamically. The first parameter specifies the index at which to replace the question marks in the query. In this case, the first question mark (index 1) will be replaced with the input

String, and the second question mark will be replaced with the action String. This means that the format of the query to be run does not to be be repeatedly specified.

- 15: Finally, the update query is executed, and as previously any SQL exceptions are handled.

5.2.5 Implementation of the Rule Engine

This class encapsulates much of the logic contained in the RuleMapper class, in order to achieve the functionality for generating the rule base, and selecting appropriate rules from the data source. It's methods include:

- selectAction(String input): Selects the appropriate action based on the input parameter.
- deleteRuleBase()
- writeRules(AbstractIndividual individual). Saves a rule base to the data source based on the genes of the Individual passed in. This should correspond to the best individual produced during the learning process.

Action Selection:

The return type for this method is String, which corresponds to the action selected based on the input parameter.

```
1  public String selectAction(String input)
2  {
3      // Run the select query from mapper object
4      HashMap<String, String> rules =  mapper.selectQuery();
5
6      String action = "";
7      //Loop over each selected rule
8      for(String key: rules.keySet())
9      {
10         System.out.println(key);
11
12         if(!compareStrings(input, key))
13             continue;
14         //if the rule condition matches the input rule is selected
15         action = rules.get(key);
16         break;
17     }
18
19     return action;
20 }
```

Fig. 23 Code Extract of RuleMapper.java

- The mapper class variable represents an instantiated object of the RuleMapper class. The

selectQuery method is called, storing every rule in the database to local variables.

- Using a for loop, each input string in the database is compared to the input string parameter in order to find an appropriate rule. If the latter occurs, the corresponding action is returned.

Saving a Rule:

This method has a return type of void, and requires an AbstractIndividual parameter, which should refer to the fittest individual generated by the Evolutionary Algorithm.

```
1  public void writeRules(AbstractIndividual individual)
2  {
3      if(mapper.selectCount() > 0)
4          mapper.deleteQuery(); //Delete old rule base
5
6      BaldwinianIndividual individualToAdd = (BaldwinianIndividual) individual;
7
8      int index = 0;
9      //Loop over each gene
10     while(index < individual.getGenes().length)
11     {
12         int ruleIndex = 0;
13         String ruleToAdd = "";
14         //Select each rule in the individual
15         while(ruleIndex < individualToAdd.getRuleLength())
16         {
17             ruleToAdd += individualToAdd.getGenes()[index];
18             index++;
19             ruleIndex++;
20
21             if(ruleIndex == individualToAdd.getRuleLength() - 2)
22                 ruleToAdd += " : ";
23         }
24         //Save the rule to the database
25         mapper.insertQuery(ruleToAdd);
26     }
27 }
```

Fig. 24 Code Extract of RuleMapper.java

- 3 – 4: The current rule base is deleted.
- 10: Each individual gene index is looped over
- 15: Each rule contained within the individual is selected
- 21: A colon is inserted, representing the separation between the input and action strings of the rule-based.
- 25: The rules is saved to the database through the RuleMapper class.
- The loop continues to the next rule.

6. Results and Testing

The results provided in this section describe the outcomes of running and comparing the three algorithm variations, using data graphs and statistical tests. The aim is to provide empirical evidence as to which variation is the most suitable for the application.

6.1 Experiments

The graphs shown here track the best and mean fitness which are produced by the algorithms, over 2000 iterations and averaged over 10 runs. Comparing the results of different runs allow some of the learning parameters such as the mutation rate to be optimized. Modifying these can have a significant impact on the algorithm's progress over the generations, and more importantly, the end result. Generally speaking, the crossover operator simply allows the population to remain diverse, whereas most population improvements will be made during mutation. The crossover rate should remain between 60% and 90% throughout the experiments, in order to ensure this. The optimal mutation rate varies strongly between different variations, meaning experimentation is required. The following graphs use a 2-dimensional plan to represent the evolutionary process; the Y axis represents a fitness value, and the X axis represents program iterations. The learning data is made of 40 data entries, meaning the optimal fitness in these cases is 40.

6.1.1 Experiment 1

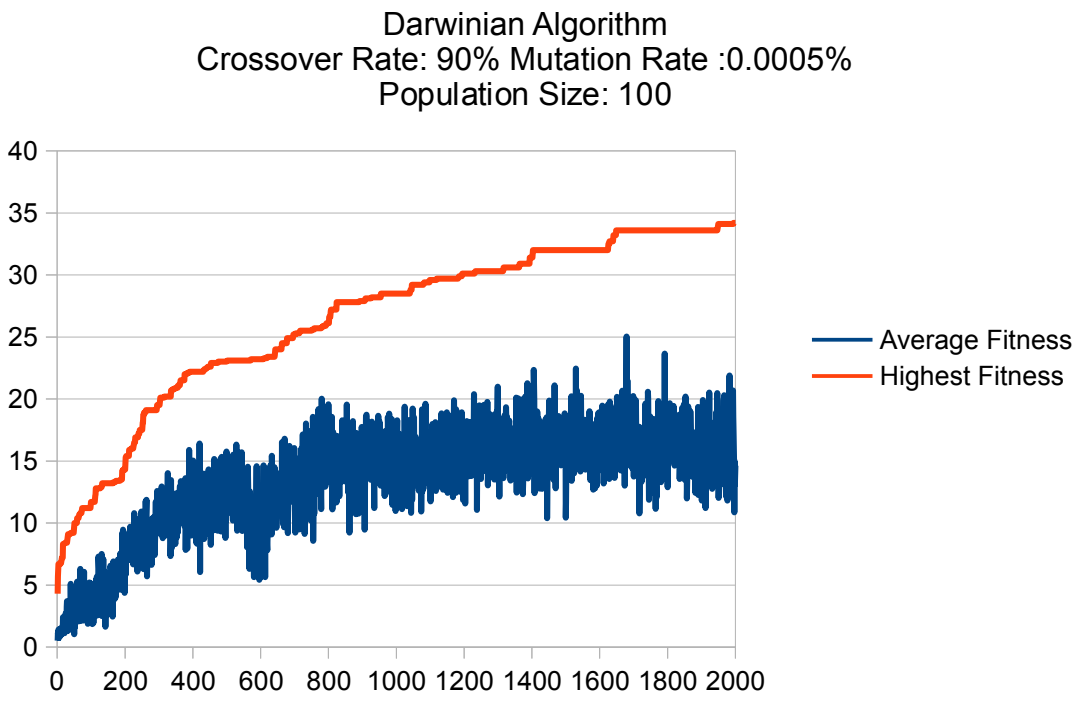


fig. 25 Darwinian Algorithm experiment 1

With a crossover rate of 90% and a mutation rate of 0.0005%, the Darwinian variation achieves successful learning. On average, by 2000 generations the algorithm has produced a best fitness of 35, and 30% of the time the maximum fitness has been achieved. Furthermore, an optimal solution has been found which can successfully classify all of the learning requirements, which amount to 40 distinguished cases, within 6 rules. Firstly, we can observe the fact that groups of solutions have a tendency to be neighbours in the search space, meaning their hamming distance is not significantly large. This is shown by sudden increases in the best solution of the population, as a slight change in it's genetic structure could result in a single rule covering several lines of the learning data. Furthermore, the shape of the previous graph suggests that by 2000 generations, the algorithm could still achieve further learning, as the population has not converged towards a single result. This would result in an evident plateau, which demonstrates a stagnating best fitness, meaning fitter solutions are unlikely to be found. This is shown in the following graph.

6.1.2 Experiment 2

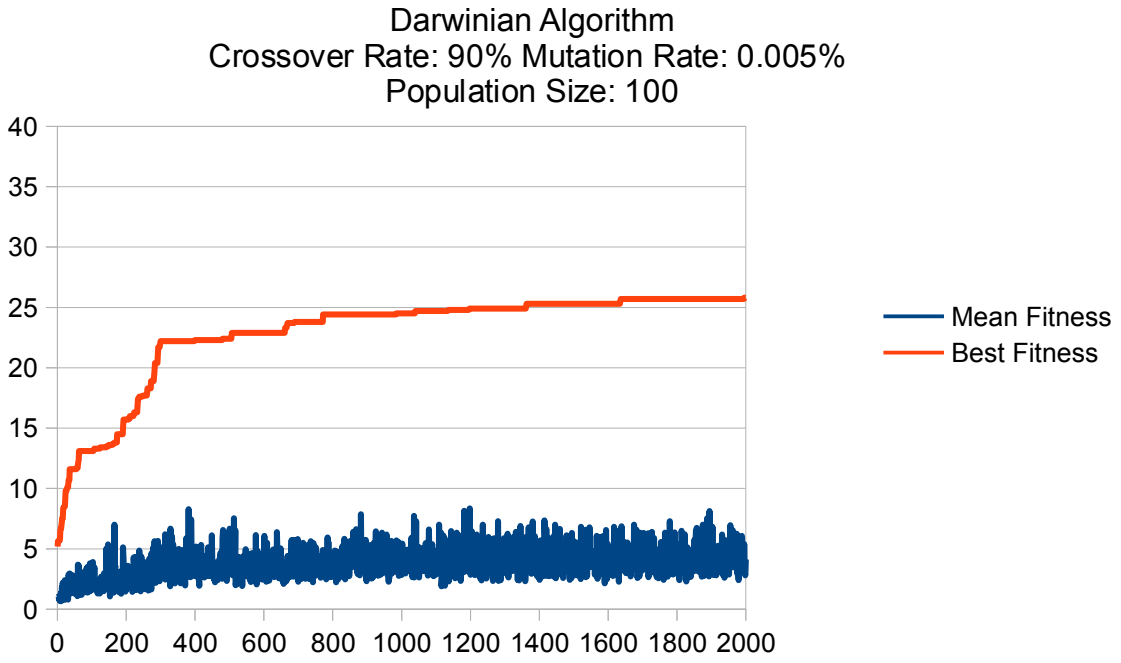
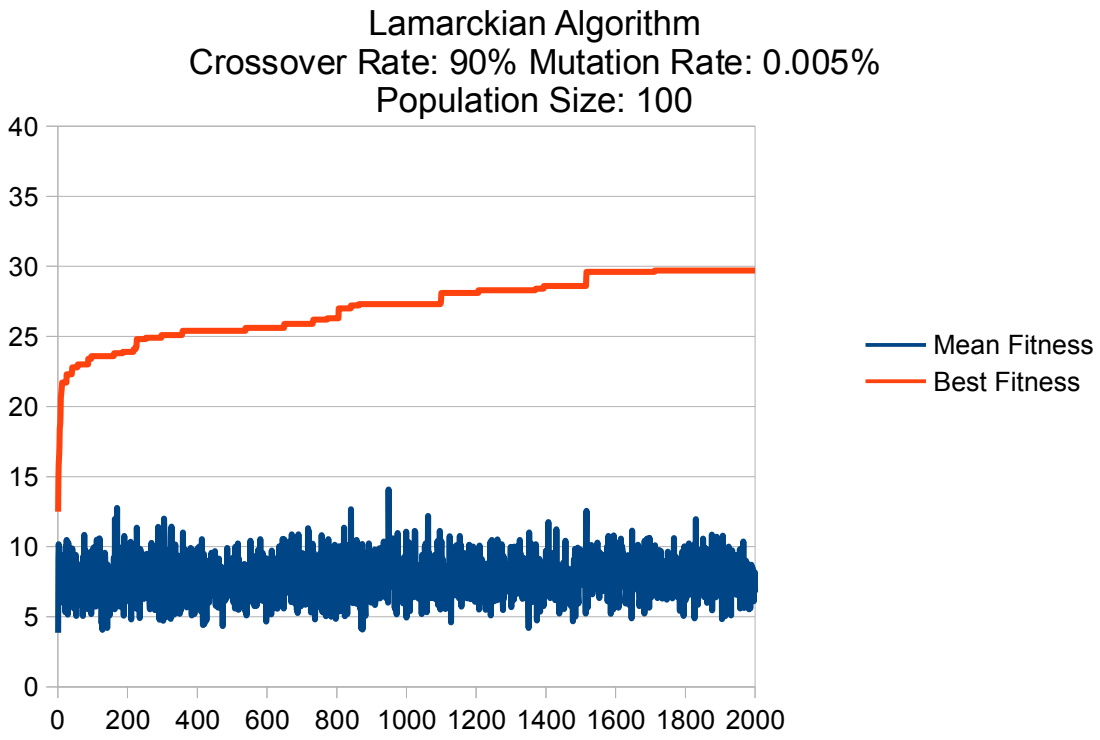


fig. 25 Darwinian Algorithm experiment 2

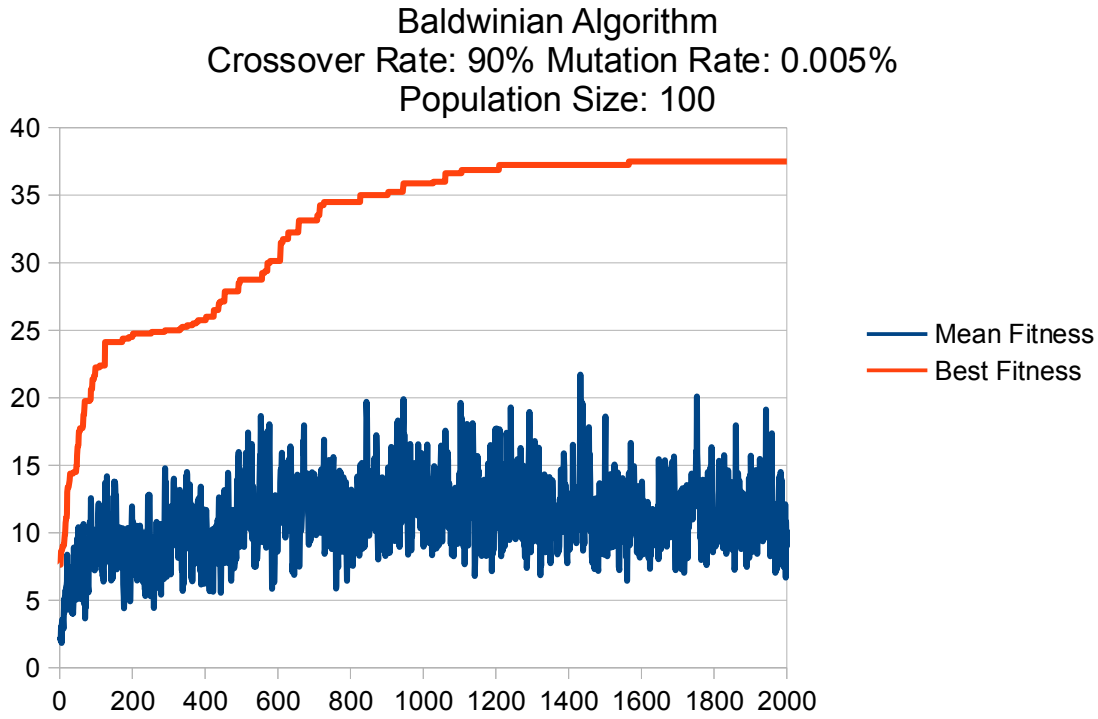
The previous experiment is unsuccessful; the initial learning pace is rapid, but tends to stagnate after 400 generations. On average, the best produced solution by the 2000th generation is undesirable. This is due to a significantly higher mutation rate, 0.005%. This means that individuals do have larger mutations applied to them on average, resulting in potential loss of portions of genes which contribute towards a high fitness. Furthermore, as the problem at hand is not linearly separable, fit solutions might be located in an area of the search space which also contains unfit solutions, meaning that a very subtle change of an individual's genes could lead to a drastic fitness increase in the mean fitness. This is not evident in the previous graph, in which the mean fitness remains stagnant throughout the process. This implies that individual's undergo too much mutation for the learning process to be incremental. Therefore a small mutation rate should be preferred.

6.1.3 Experiment 3



The Lamarckian variation does not achieve the same results as the initial experiment. Using the same learning parameters, the population's average fitness does not progress whatsoever, and the best individual produced seems to stagnate, on average, at a fitness of approximately 30. As the local search algorithm is applied to 90% of individuals, it has a similar effect as using a high mutation rate in the previous experiment; individuals undergo too much change throughout their life cycle, meaning incremental changes cannot be applied, and the population converges prematurely.

6.1.4 Experiment 4



The Baldwinian algorithm apparently produces the most significant results. Similarly to the Darwinian variation, it has produced an average mean fitness of 35, however the learning has been achieved in approximately half the amount of generations. This variation has a more sophisticated and complex mechanism. As the adaptability of an individual is a parameter which is evolved throughout the process, solutions with a high probability of adapting will display large increases in fitness, whereas solutions with a lower adaptation probability will allow them to be fine tuned when required. This concept is known as exploration and exploitation.

The previous graphs would suggest that the Baldwinian is the most suitable for our application, the Lamarckian being the least efficient. However, we need to determine whether the Baldwinian algorithm statistically produces improved results over the other two. We can only infer from the previous graph that the Baldwinian variation is the best solution. Through a Student's t-test, we can offer empirical evidence as to which algorithm will consistently provide the best learning rate.

6.2 Statistical Tests

6.2.1 The Student's T-Test

Once the separate algorithms have been implemented, we will need a means of determining which is the most suitable to our application. The Student's T-test (Hair, 1992) is a widely used technique for comparing two data sets. Essentially, a T-test provides the means for determining the chances that two data sets will be consistently different, or not, in any future tests. The hypothesis that two different data sets are significantly different is tested. This provides an alternative to simply comparing limited data, which will not prove significant differences. A T statistic is calculated by comparing the variance between the two data sets to the variance within the data sets. The formula for calculating a Student's T-test is:

$$t = \frac{\mu_1 - \mu_2}{SE\,\mu_1\mu_2}$$

Where u1 and u2 represent the mean of each data set, and SE represents the Standard Error of difference in the group means.

If the returned value is large enough, then the two data sets are considered statistically different, and not just different in the samples compared.

However, most programs which calculate a T value such as Microsoft excel, return a value known as a P value, and not the t statistic. The P value simply measures the chances of obtaining the same calculated t value with random data. The Pvalue threshold is usually set a 0.05, meaning that the chances of producing the same results with random data should equal to or less than 5%. Any t test which produces a p value of 0.05 or below can be considered successful, meaning the two data sets are in fact reliably different.

6.2.2 Experiment 1

In a new experiment, each algorithm is executed a number of times, saving the fitness of the best solution which has been produced at the final iteration. This needs to be repeated enough times to produce statistically consistent data. The following table shows the resulting P value of a two tailed T-test, using data obtained after 2000 iterations of each algorithm over 100 runs. A two tailed T-test simply means that the two data sets originate from different sources.

Algorithm Pairs	P value	Superior Results
Darwinian vs Lamarckian	0.016	Darwinian
Darwinian vs Baldwinian	0.053	NA
Lamarckian vs Baldwinian	0.011	Baldwinian

The previous results are inconclusive; The only implication which can made is that the Darwinian and Baldwinian algorithms have a high probability of producing consistently different results to the Lamarckian algorithm, over an infinite amount of runs. We can infer that the Baldwinian and Darwinian algorithms are superior to the Lamarckian algorithm. In order to determine the more efficient algorithm of the Darwinian and Baldwinian algorithms, we need to run a new experiment, this time using only 1000 iterations of each algorithm as a single run.

6.2.3 Experiment 2

The returned p value is 0.0061, meaning that the two data sets are now statistically separable. The result of this experiment combined with the results shown in the graphs produced previously would strongly suggest that the Baldwinian variation is the most suitable for our application.

6.3 Software Testing

In the final phase of the Waterfall development process, system tests should be carried out. It is widely believed that the aim of testing software applications is to prove that the latter functions as it is intended to. In reality, it aims to discover any bugs which exist within the software, which is subtly distinct (Myers, 2011). It is sensible to assume that writing large systems will result in a certain amount of errors, and writing tests which do not uncover these should be considered unsuccessful. Testing does not involve fixing program errors, but is merely used to uncover and understand them. Three forms of testing must be considered for out system. A full test plan is provided as an appendix in this report.

6.3.1 Unit Testing

Unit tests are designed to test the logic contained within a single method or module. They do not test interactions between classes and objects, which is covered by integration tests. Unit tests should define example input to be passed to modules, in addition to the expected output or event produced when the input is applied. An assertion is then made as to whether the actual output is equal to the expected output, which determines whether the test has passed. It involves writing a test which asserts the validity of a unit or component of the software, prior to writing the code itself. If the code written passes the test, the module is valid.

6.3.2 Integration Testing

Integration tests are designed after unit tests, and group individual modules of a system to perform wider testing of core functionality of the system. Testing the integrity of a Service Oriented Architecture involves verifying that the interactions between various layers of the system are consistent. For example, and integration test should be written to verify that the mapper layer transfers data consistently between the domain layer and the database layer.

6.3.3 System testing

System tests are a form of black box testing which do not take into account any technical details of the software's implementation. These allow testers to verify that the initial requirements of the software have been met.

7. Conclusions and Future Work

Extensive research has been made into the functionality of prediction and classification algorithms, particularly in the domain of Evolutionary Algorithms. Following a Waterfall development model, we have successfully outlined the necessary phases which need to be carried out in order to design a predictive server scaling system. While some aspects of the system such as the Client-Server distinction and the Service Oriented Architecture are absolutely fundamental for the functionality and stability of the software other aspects, notably the prediction algorithm and cloud provider, are interchangeable. One of the main intents of this project was to design a system which is entirely host independent, allowing businesses to integrate this system with their technologies of choice.

We have shown here that Evolutionary Algorithms are a perfectly viable means of classifying cloud server scaling actions. The combination of algorithm results and statistical testing has proven that the Baldwinian Memetic Algorithm is the most suitable of the three implementations, although the Darwinian Genetic Algorithm is also appropriate. The Lamarckian Memetic Algorithm should be avoided within the context of this project, as it's inherent characteristic of rapid convergence is not suited to the requirements of the problem.

The original plan for this project was to implement a Learning Classifier System as the prediction and classification algorithm for the software. However due to time constraints and technical requirements for the LCS, this was shortly changed to the use of simple Evolutionary Algorithms . With additional time, the LCS implementation would be extremely beneficial, as it combines both prediction and action mechanisms, and proves to be extremely versatile when interacting with a non static environment. Furthermore, UML sequence and Deployment diagrams could be integrated with the rest of the system design, showing the software's behaviour over time.

A slight shortcoming of the project was the failure of an initial plan to compare a Michigan style representation strategy, based on the individual design described in the original LCS, to the Pittsburgh strategy which has been employed. The Pittsburgh approach proved to be perfectly adequate, although a comparison of the two approaches could have yielded some interesting results.

REFERENCES

Journals and Books:

Abidi, F.A. and Singh, V.S. (2013) Cloud Servers VS. Dedicated Servers – A Survey. *Innovation and Technology in Education (Mite), 2013 IEEE International Conference in MOOC.*, pp. 1-5.

Azar, A.T. and El-Said, S.H. (2013) Probabilistic neural network for breast cancer classification. *Neural Computing and Applications*. 23 (6)

Budgen, D.B. (2003)*Software Design*. 2nd ed. Harlow: Pearson Addison Wesley.

Bull, L. (2005) Two SImple Learning Classifier Systems. *Studies in Fuzziness and Soft Computing*. 183

Butz, M.V. (2006) *Rule-based Evolutionary Online Learning Systems : a Principled Approach to LCS Analysis and Design*. Berlin: Springer.

Chakraborty, U.K. and Janikow, C.Z. (2003) An analysis of gray versus binary encoding in genetic search. *Information Sciences*. 156, pp. 253-269.

Coulouris, G. (2012)*Distributed Systems : Concepts and Design*. 5th ed. Harlow: Addison-Wesley.

Crichlow, J.M. (2000) *The Essence of Distributed Systems*. 1st ed. Harlow: Pearson Education Limited.

Digalakis, J. and Margaritis, K. (2004) Performance comparison of memetic algorithms. *Applied Mathematics and Computation*. 158 (1), pp. 237-252.

Eiben, A.E. (2003) *Introduction to Evolutionary Computing*. Berlin; London: Springer.

Emile, A. (2003) *Local Search in Combinatorial Optimization*. Princeton N.j: Princeton University Press.

Finlay, P. (1994) *Introducing Decision Support Systems*. 2nd ed. : Ncc Blackwell.

Fowler, M. (2003) *Patterns of Enterprise Application Architecture*. 1st ed. Boston, Mass. ; London: Addison-wesley.

Gançarski, P. and Blansché, A. (2008) Darwinian, Lamarckian, and Baldwinian (Co)Evolutionary Approaches for Feature Weighting in K-MEANS-Based Algorithms. *Ieee Transactions on Evolutionary Computation*. 12

21: Garg, P. (2009) Comparison between memetic algorithm and genetic algorithm for the cryptanalysis of simplified data encryption standard algorithm.*International Journal of Network Security & Its Applications*. 1

Goldberg, D.E. (1989) *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading: Addison-wesley.

Grady, J.O. (2006) *System Requirements Analysis*. Amsterdam; London: Academic.

Hair, J.F. (1992) *Multivariate Data Analysis with Readings*. 3rd ed. : Macmillan.

Hart, W.E., Krasnogor, N. and Smith, J.E. (2005) Memetic Evolutionary Algorithms. *Recent Advances in Memetic Algorithms*.

Hastie, T. (2001) *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. 1st ed. New York: Springer.

Holland, J.H. (1973) Genetic algorithms and the optimal allocation of trials.*Siam Journal of Computing*.

Holland, J.H. (1976) Adaption. *Progress in Theoretical Biology*.

Ishibuchi, H., Nakashima, T. and Murata, T. (1997) Comparison of the Michigan and Pittsburgh Approaches to the Design of Fuzzy Classification Systems.*Electronics and Communications in Japan*. 80 (12)

Khengab, C.W., Chongb, S.Y. and Lim, M.H. (2012) Centroid-based memetic algorithm – adaptive lamarckian and baldwinian learning. *International Journal of Systems Science*. 43 (7)

Maruyama, O. (2013) Heterodimeric protein complex identification by naïve bayes classifiers. *Bmc Bioinformatics*.

Michael, M., Moreira, J.E., Shiloach, D. and Wisniewski, R.W. (2007) Scale-up x Scale-out: A Case Study Using Nutch/Lucene. *Parallel and Distributed Processing Symposium, 2007. Ipdps 2007. Ieee International*.

Myers, G.J. (2011) *The Art of Software Testing*. : John Wiley & Sons.

Norton, R.L. (2002) Using virtual Linux servers. *Ieee Computer*.

Patterson, D.W. (1996) *Artificial Neural Networks: Theory and Applications*. : Prentice Hall International.

Petridis, V., Kazarlis, S. and Bakirtzis, A. (1998) Varying fitness functions in genetic algorithm constrained optimization: the cutting stock and unit commitment problems. *Ieee Transactions on Systems, Man, and Cybernetics—part B: Cybernetics,*. 5

Picton, P. (2000) *Neural Networks*. Basingstoke: Palgrave.

Rothlauf, F. (2006) *Representations For Genetic and Evolutionary Algorithms*. 2nd ed. Berlin: Springer.

Sinan Si, A. (1998) *UML in a Nutshell : a Desktop Quick Reference*. Beijing: O'Reilly.

Smith, J. (2005) The co-evolution of memetic algorithms for protein structure prediction. *Recent Advances in Memetic Algorithms*.

Van Laarhoven, P.J.M. (1987) *Simulated Annealing : Theory and Applications*. Dordrecht: Reidel.

Wang, H., Wang, D. and Yang, S. (2009) A memetic algorithm with adaptive hill climbing strategy for dynamic optimization problems. *Soft Computing*.

Websites:

Amazon (2014) *Amazon Web Services.* Available From: *http://aws.amazon.com/documentation/ec2/*

Microsoft (2014) *Microsoft Azure.* Available From: http://azure.microsoft.com/en-us/documentation/

Netflix (2013) *The Netflix Tech Blog.* Available From: http://techblog.netflix.com/

Rackspace (2014) *Rackspace The Open Cloud Company.* Available From: http://docs.rackspace.com/servers/api/v2/cs-gettingstarted/content/overview.html

APPENDIX A – Test plan

1. Unit Tests

Gray Encoder Test Plan:

Tested Method	Input/Expected Output	Input/Output Data Type
Gray Encoder	1111 → 1010 1011 → 1101 0111###1101## → 0100###1011##	String, String
Gray Decoder	1010 → 1111 1111 → 1000 0100###1011## → 0111###1101##	String, String

Selection Test Plan:

- Instantiate population of 50 Individual
- Randomly assign each individual a fitness value between 0 and 25, to attempt a normal distribution of possible fitnesses across all individuals
- Run the selection method 100 times and calculate the percentage of selections for the fittest individual
- Repeat this step 100 times with a different population each time in order to calculate a total average
- This average should be approximately 20% for tournament selection.

Crossover Test Plan:

- Instantiate a population of two individuals, with 10 genes each
- Run crossover with a 50% crossover rate, calculate the hamming distance between the new and old individuals
- Repeat this process 100 times to calculate an average
- This should be approximately 5.

Mutation Test Plan:

- Instantiate a single individual with 100 genes.
- Run 100 mutations for the same individual, with a mutation rate of 0.1
- The total amount of mutations should be slightly less than if not equal to 100

Hill Climber Local Search Test Plan:

Input a ten digit binary string, and print the modified string at each iteration of the algorithm.
At any given iteration, the output string must only have a hamming distance of 1 from the previous output.
Furthermore, the output String at each iteration must always be unique to a single run of the algorithm

Example Input: 1##110#10
Example Output (Modifications are random)

0##110#10
10#110#10
1#1110#10
1##010#10
1##1#0#10
1##11##10
1##110010

Fitness Calculator Test Plan:

- Define a number of server scaling objectives
- Create a new Individual object, and set it's genes variable to be exactly equal to these objectives.
- Using a FitnessCalculator object, calculate the fitness of the previously instantiated Individual object.
- The returned fitness should be exactly equal to the number of objectives.
- Similarly, to test the penalty based system, a number of constraints should be defined
- Create a new Individual object, and set it's genes variable to be exactly equal to these constraints. Set the object's fitness to be exactly equal to the number of defined constraints
- Using a FitnessCalculator object, calculate the Individual's fitness.
- The returned fitness should be equal to 0.

2. Integration Tests

Database Integration Test Plan:

- Generate a rule base of several rules using any of the classes which extend AbstractAlgorithm.
- Store the result of the previous operation in a local String variable.
- Call the insertQuery method on a RuleMapper object, passing an Individual object as parameter.
- Call the selectQuery method on the same RuleMapper object, and store the result in another String variable. This should equate to the genes of the previously inserted Individual object.
- Perform an assertion on the two strings, which checks that they are exactly equal.

3. System Tests

Test Plan:

- A rule base is generated by the user through a manual run of each Evolutionary Algorithm.
- Multiple test inputs are designed by the user, which should amount to at least 15 input strings which correspond to each action type – 5 per action.
- The user verifies whether the correct scaling action API call has been made for each input