Imperial College London
Department of Computing

Exploring the Potential of a Novel Time Series Prediction Algorithm

Josh Forman-Gornall

June 2013

Supervised by Dr. William Knottenbelt

Submitted in part fulfilment of the requirements for the degree of
Master of Engineering in Computing of Imperial College London

Abstract

We investigate the performance and applicability of an innovative and novel time series prediction technique, which differs greatly from conventional statistical and machine learning techniques such as ARIMA and artificial neural networks. This technique is a powerful, online and non-parametric learner. Details of the underlying algorithm are protected by an NDA.

We initially develop an implementation of the technique which includes an intricate 3D visualisation of the underlying models and provides statistical functions and utilities which can be used to analyse the predictive power of the technique. We present a thorough analysis of the complexity and accuracy of these models tested on a range of artificial and real-world time series, with a strong emphasis on non-linear chaotic time series which are known to be very difficult to predict. Benchmark chaotic systems such as the Mackey-Glass series and the Lorenz system are among those analysed. The predictor's ability to handle noisy data is also studied. The predictions made by the novel technique are compared with those made by current state-of-the-art machine learning techniques - in particularly the multi-layer perceptron and the support vector machine.

We find that the algorithm generally constructs good models which capture the periodicity of periodic series and the determinism in chaotic systems. Our experiments show that the algorithm is often able to give more accurate predictions than a basic multi-layer perceptron or support vector machine - however this is dependent on the series. The algorithm's ability to perform single-step prediction is particularly strong, whereas we find its multi-step prediction capability to be limited and problematic. In the presence of noisy data, we find that the models produced are not always optimal. When compared with the results of recent literature we find that the algorithm performs better than statistical methods such as exponential smoothing and ARIMA, however adaptations of the neural network model can outperform the algorithm in some cases.

Acknowledgements

I would like to thank Dr. William Knottenbelt for his continuous support and enthusiasm towards this project. I would also like to thank Ben Rogers, firstly for coming forward with the idea on which this project is based and also for his time, effort and inspiration.

Most of all, I would like to thank my parents - for putting my education first when I was in school and for supporting me at university. Without them, I would not have come this far.

Contents

1 Introduction

In this project, we investigate a time series prediction technique devised by Ben Rogers, a programmer with an interest in machine learning and chaotic series. The underlying algorithm differs from conventional time series prediction methods in that it is not based on statistical methods or machine learning techniques; it is a completely novel approach which to the best of our knowledge has never been attempted before. With a completely unknown potential, the algorithm provides scope for an interesting investigation - and in this project we explore the applicability and power of the algorithm with the aim of discovering its strengths and weaknesses.

1.1 Motivation

The analysis of time series data with a view of forecasting future values in the sequence is known as time series prediction. This is a significant and challenging task in machine learning which has a diverse range of real-world applications, such as using stock price movements to predict the direction of financial markets [1][2], predicting the weather [3] and even forecasting an earthquake from the analysis of historical seismic activity [4].

Prior to beginning the project, the prediction technique had already been demonstrated to work effectively for several periodic time series including the sawtooth and sine waves. However, most intriguingly, it has shown the capability to learn complex aperiodic mathematical time series, such as the chaotic logistic map.

Many of the most well established time series analysis techniques, discussed in Section 2.2 are parametric, meaning they make assumptions about the structure of the underlying stochastic process of the time series. In order to make these assumptions, analysis of the time series has to be undertaken and parameters describing the structure must be passed to the learning algorithm. This manual process can be very difficult and time consuming.

In contrast, the technique we investigate in this project is non-parametric and no assumptions need to be made about the structure of the time series. We can simply pass an arbitrary time series to the algorithm and it will begin constructing a predictive model immediately. In addition to this, the algorithm is an online learner - meaning it learns and refines its model with every observation, in real-time. If a technique with these characteristics is able to match the performance of established parametric methods, then this could prove to be very powerful and influential in the field of time series analysis. Reflecting the iterative algorithmic nature of the approach, in this report the technique will be referred to as the ALgorithmic Prediction Engine (ALPE).

1.2 Objectives

This is an exploratory project in which our primary objective is to investigate and discover the potential of this novel time series prediction technique. The existing implementation of the algorithm is written in BASIC, one of the first high-level computing languages. The language is not the most suitable choice for an investigation as it is difficult to read and does not benefit from the advantages of object-oriented programming. Further limitations of the existing implementation are that it is only able to work with 8-bit integers and that it is only able to make single-step predictions (i.e. predicting only one time-step in the future). With this in mind, the key objectives of this project are to:

- Develop an implementation of the algorithm in an object-oriented programming language which can work with an arbitrary number of bits and make multi-step predictions. The implementation must provide an API which is easy and understandable for any programmer to use without requiring an understanding of the underlying algorithm.
- Apply the algorithm to various synthetic and real-world data sets and investigate the complexity and accuracy of the predictive models produced by the algorithm.
- Compare the algorithm's performance with that of established techniques. We aim to find out in which cases the algorithm has potential

to outperform these techniques and also to discover any flaws or weaknesses in the algorithm.

- Explore the possibility of applying the algorithm to problems outside the domain of time series prediction.

1.3 Contributions

This project has a heavy focus on both software development and the analysis of time series prediction techniques. The main contributions are as follows:

- An easy to use and extensible C++ implementation of ALPE which provides a graphical display of progress and statistics which are updated in step with the algorithm.
- An API which provides the functionality to interact with the underlying prediction algorithm - triggering predictions and advancing the predictor's learning process.
- A stunning 3D visualisation of predictive models.
- Statistical functions and utilities which can be used to analyse the algorithm's power and applicability.
- Analysis of the algorithm's performance when applied to simple, periodic and chaotic time series - including both artificially generated series and those derived from real-world measurements.
- A comparison of the algorithm's performance with that of established and emerging techniques. Includes comparisons with statistical techniques (e.g. ARIMA), machine learning techniques (e.g. MLPs and SVMs) and techniques developed in recent research papers.
- Exploration of the potential to apply the algorithm to compression problems - including development of a compression analysis utility and analysis of compression capability for ASCII text files.

1.4 Report Structure

This report is further organised as follows:

- Chapter 2 gives an introduction to time series, including key properties as well as a description of relevant chaotic series. An outline of established prediction techniques and a summary of recent research is also provided to aid the understanding and interpretation of this report.
- Chapter 3 describes and justifies the tools and libraries used in the creation of the C++ implementation. An overview of the class layout and a description of the functions made available by the API is also provided.
- Chapter 4 provides implementation details and presents an overview of the key features and extensions to the basic implementation.
- Chapter 5 focusses on the experimental aspect of the project. Results are presented on the analysis of the models along with a detailed and quantitative comparison of ALPE with established techniques and emerging techniques developed in recent literature.
- Finally, Chapter 6 presents a summary of our results and highlights the strengths and weaknesses of ALPE.

2 Background

2.1 Properties of Time Series

A *time series* is an ordered sequence of values taken from a variable observed at equally spaced time intervals. In this project we deal with univariate time series - those which consist of a single scalar observation at each point in time.

2.1.1 Autocorrelation

All time series considered in this project exhibit the property of *autocorrelation*. This means that successive values in the series are not independent; they depend on each other. Any time series which does not have this property is random, and for such series it would not be possible to predict future values.

To examine the autocorrelation of a data set, we can plot the *autocorrelation function* (ACF) which shows the correlation of the data for various lags. For example, for a lag of one the ACF would determine the correlation between observations at an arbitrary time t, X_t and the observation one time period earlier, X_{t-1}. Figure 2.1 shows an example plot of the autocorrelation for a sawtooth wave. In this plot the blue lines indicate bounds for statistical significance, and we can see that there are significant correlations for many lags. This is expected as a sawtooth wave is a simple repeating pattern in which there are bound to be correlations between future values and past values.

2.1.2 Stationarity

In time series models, stationarity is an important assumption as it implies that the series behaves in a similar way regardless of time - meaning its statistical properties are constant over time. A series has the property of

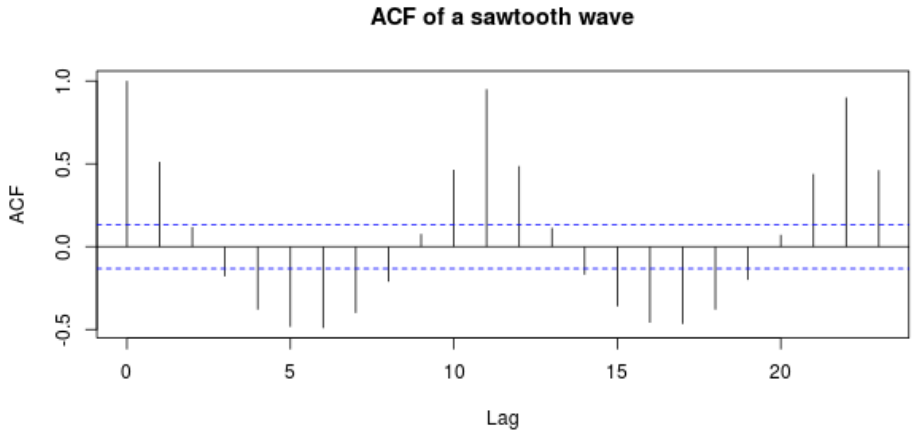

Figure 2.1: The autocorrelation function of a sawtooth wave

stationarity if it is without trend, has a constant autocorrelation structure and no periodic fluctuations. This type of series has a mean, variance and autocorrelation which do not vary over time.

2.1.3 White Noise

A series of observations, $\{\epsilon_t\}$ is *white noise* if its elements are independent and identically distributed random variables with a mean of zero. In the case where ϵ_t is normally distributed with zero mean, the series is called *Gaussian white noise*. Equation 2.1 shows the expected output of the autocorrelation function on white noise.

$$ACF = \begin{cases} 1 & \text{if lag } = 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

In time series prediction, the differences between our predicted values and the actual observations are called *residuals*. The concept of white noise is very important in time series prediction, because in any accurate model of a time series the residuals must be white noise. If this is not the case, then our learned model is missing structure in the time series. Analysis of the residuals can be performed by plotting their autocorrelation and partial autocorrelation functions. The ACF plot for the residuals should

look similar to that of figure 2.2, where there is no significant correlation for any lag.

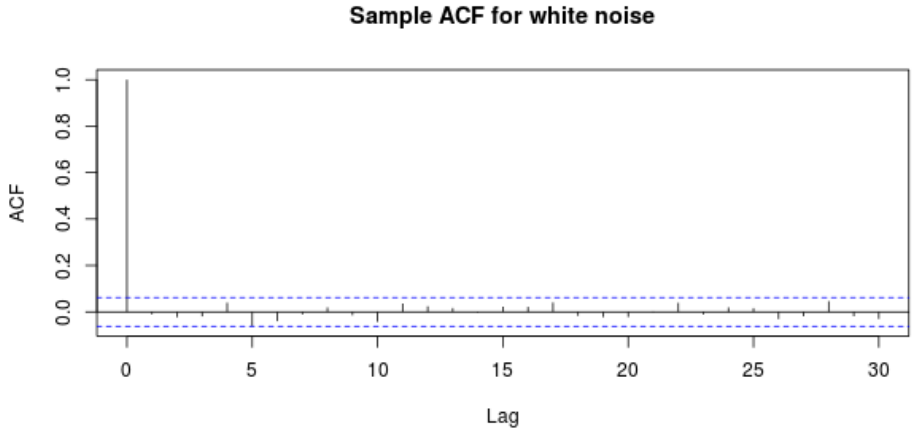

Figure 2.2: The sample autocorrelation function of white noise

2.2 Current Forecasting Techniques

This section presents some established techniques which are used to understand and predict future values in the series. These techniques can be used for both single-step and multi-step prediction. Single-step prediction is where we predict the next value in a time series, whereas in multi-step prediction, multiple future values are predicted using the observations made so far.

2.2.1 Autoregressive Models

An *autoregressive* (AR) model is a linear regression of the current value of the time series against previous values in the series. An AR model which uses p previous values to forecast future values is said to have order p and is denoted AR(p). In this case an observation at time t can be written:

$$X_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + ... + \phi_p X_{t-p} + \epsilon_t + c \tag{2.2}$$

where $\phi_1, ..., \phi_p$ are the parameters of the model, c is a constant and ϵ_t is white noise.

2.2.2 Moving Average Models

Moving average (MA) models forecast future values based on a linear combination of past forecast errors (i.e. noise). An MA model which considers the error of the last q observations is said to have order q and is denoted MA(q). In these models, an observation at time t can be written:

$$X_t = \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + ... + \theta_q \epsilon_{t-q} + \epsilon_t + \mu \tag{2.3}$$

where $\theta_1, ..., \theta_q$ are the parameters of the model, μ is the mean of the series (often assumed to be zero) and ϵ_{t-i} are white noise.

2.2.3 ARMA Models

The AR and MA models can be combined to give a sophisticated forecasting technique: the *autoregressive moving average* (ARMA) model. This model considers both past observations and past errors when predicting future values. ARMA was largely developed by two statisticians, George Box and Gwilym Jenkins [5] and is sometimes referred to as the Box-Jenkins model. An ARMA model which considers the p previous values and the q previous errors is denoted ARMA(p,q). An observation at time t is then given by:

$$X_t = \sum_{i=1}^{p} \phi_i X_{t-1} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \epsilon_t + c \tag{2.4}$$

A method called differencing can be applied to convert a non-stationary time series into a stationary time series.

2.2.4 ARIMA Models

In many cases the time series in question consists of non-stationary data which is not suitable for the standard ARMA model. An extension of ARMA is the *autoregressive integrated moving average* (ARIMA) model which addresses this problem. In ARIMA, a technique called *differencing* is applied to the data as a preliminary step. Equation 2.5 shows how this technique can be applied to give the first order difference of a time series, D_t.

$$D_t = X_t - X_{t-1} \tag{2.5}$$

Once a stationary series has been obtained, the ARMA method is then applied and the results are summed (or *integrated*) to give a final prediction.

The Box-Jenkins models have been applied successfully in a wide variety of situations. For example, they have been used in prison planning [6] and to forecast wheat production in Pakistan [7]. While these models are clearly very successful, they can be tedious and time consuming to use as the preliminary steps of model identification, parameter estimation and model diagnosis need to be carried out before any predictions can be made. These models are therefore not suitable for making real-time predictions from high frequency data. They are also limited by the assumption of a linear form for the model. As a result of this, ARIMA models are unable to capture the highly non-linear relationships which are found in many chaotic and real-world time series.

2.2.5 Machine Learning Methods

Another approach to time series forecasting is to use techniques from the fields of machine learning and data mining. This ranges from simple methods such as linear regression, to more powerful techniques such as the Multi-Layer Perceptron (MLP) and Support Vector Machine (SVM). MLPs and SVMs are often applied to forecasting problems, as they are capable of learning and modelling the non-linear trends in data.

Before these techniques can be applied, the temporal ordering must be first removed from the time series. However, the temporal information must be preserved in some way. This can be done by encoding the time dependency using additional input fields, sometimes referred to as *lagged variables* [8]. Once the data has been transformed there are many machine learning and data mining techniques which can be applied.

These are very different to the classical statistical techniques discussed earlier and in many cases has been shown to be a more powerful approach - for example Thissen *et al.* demonstrate how a support vector machine is able to clearly outperform the ARMA method at predicting a chaotic time series [9].

2.3 Artificial Noise

Many real-world time series have a noisy component which obscures the structure of the underlying data. This noise could arise from the observations themselves or from inaccuracies in the way observations are measured. The problem of distinguishing between noise and structure is a key obstacle for time series prediction methods. This is often a difficult task, as the precise nature (e.g. the distribution and level) of the noise is usually unknown.

When creating synthetic time series, we may wish to introduce a noise component to a known series in order to replicate the effect of noise. As discussed in section 2.1.3 the noise component should have the properties of white noise. We can therefore generate *artificial noise* by sampling from a probability distribution, as long as we ensure that our distribution has zero mean and a constant variance. Figures 2.3 and 2.4 show the effect of artificial noise on a time series given by a sine wave signal.

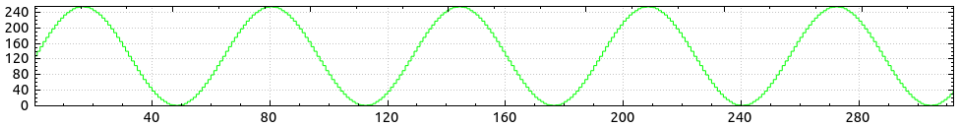

Figure 2.3: Sine wave without a noise component

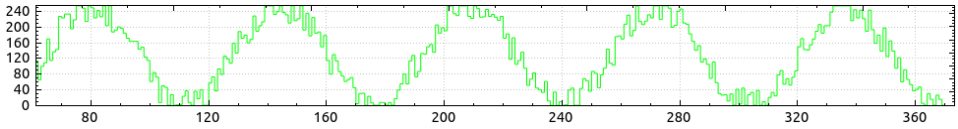

Figure 2.4: Sine wave with uniformly distributed artificial noise

In this project, we use a variant of the *Mersenne Twister* for all random number sampling. With a long period of $2^{19937} - 1$ we can be confident in the quality of our psuedorandom numbers.

2.4 Measuring Error

Comparing techniques and studying their performance is an important part of this project and so we need to be able to quantify their ability to make accurate predictions. For this purpose, we will use an error metric known as the root-mean-squared error (RMSE) as a way to score an algorithm's performance at predicting future values in a time series. With this metric, we take the square of each of our residuals and then find the square root of their mean. Equation 2.6 shows the RMSE for n predictions, where p_i and o_i signify a prediction and a corresponding observation.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(p_i - o_i)^2} \tag{2.6}$$

The advantage of using the RMSE is that larger residuals are penalised more as a result of the square, making the error significantly higher if there are large residuals. This is good for us since large errors are particularly undesirable in time series prediction.

The RMSE has the same units as the data being predicted. In order to make comparisons between models across different data sets, we require a relative measure. For this purpose, we use the normalised root-mean-square-error (NRMSE) which is independent of the units and scaling of the data. Equation 2.7 shows the NRMSE for n predictions, where p_i and o_i signify a prediction and a corresponding observation and $\overline{o}$ is the mean of the observations.

$$NRMSE = \sqrt{\frac{\sum_{i=1}^{n}(p_i - o_i)^2}{\sum_{i=1}^{n}(p_i - \overline{o})^2}} \tag{2.7}$$

An NRMSE of zero indicates a perfect prediction, while a value greater than 1 indicates that the predictions are no better than using the mean value of the time series, $\overline{o}$.

2.5 Chaotic Series

Chaos theory states that small systems and events can cause very complex behaviour or dramatic outcomes, and that these complex outcomes are actually the result of a sophisticated, underlying order [10]. In fact, a chaotic system is a purely deterministic process despite having a seemingly random appearance. The analysis of these series is very difficult because they consist of stable and unstable states, interwoven in an extremely complicated pattern [11]. Therefore, a successful technique for chaotic time series prediction must be able to capture very subtle deterministic features.

Many processes found in our world and in nature exhibit chaotic behaviour. Solar system dynamics, the economy, the weather, river flows and electrical power consumption are just a few examples of processes where chaotic behaviour can be found. A key characteristic in all of these processes is that they are highly sensitive to initial conditions. Therefore, as shown in Figure 2.5, a slight change in the initial conditions can result in a dramatically different series of observations. As a result of this, long term prediction of chaotic time series is impossible - however, highly accurate short term prediction can be achieved.

For chaotic series we can plot a *phase space diagram*, which is simply a plot of X_t against X_{t-1}. These plots represent the possible states of the chaotic system and often form distinctive shapes, showing the determinism in a seemingly random system. For example, as depicted in Figure 2.6, the phase space plot for the chaotic logistic map resembles an inverted parabola. When predicting chaotic series, our predictions can be used to plot a phase space diagram which we can compare to the expected diagram - giving us an immediate idea of how well the predictor is performing.

In this section some benchmark chaotic series are introduced, along with a discussion of previous attempts to predict them.

2.5.1 Chaotic Logistic Map

The logistic map is a very simple non-linear equation, given by Equation 2.8.

$$x_{n+1} = rx_n(1 - x_n) \tag{2.8}$$

Interestingly, very complex chaotic behaviour can be observed when the

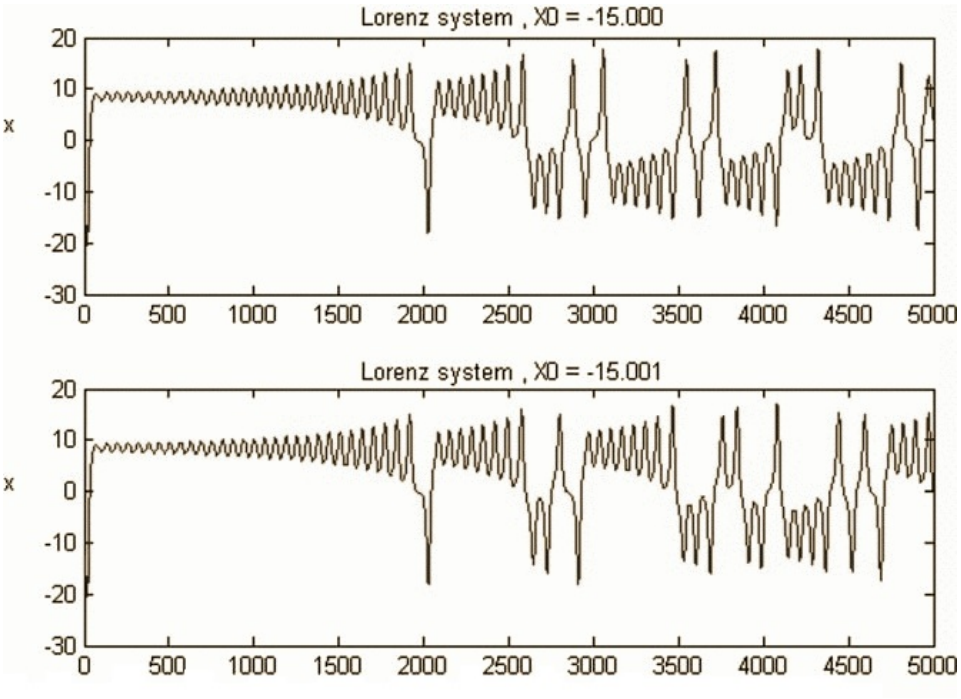

Figure 2.5: The effect of a small change in initial conditions [12].

r parameter is set close to 4. In all tests discussed in this report, we use an r value of 3.99.

2.5.2 Mackey-Glass Chaotic System

The Mackey-Glass chaotic system is given by a non-linear time delay differential equation (Equation 2.9).

$$\frac{dx(t)}{dt} = \frac{0.2x(t-\tau)}{1+x(t-\tau)^{10}} - 0.1x(t) \tag{2.9}$$

When generating Mackey-Glass data for our experiments, we will configure our parameters as follows: time step (Δt) is 0.1, the initial value (x_0) is 1.2 and $\tau = 17$. These are the same parameters as are used in [12][13] - enabling us to make meaningful comparisons.

The Mackey-Glass system has been used as a model of white blood cell production [14] and with its strong chaotic behaviour has become a benchmark for chaotic time series prediction.

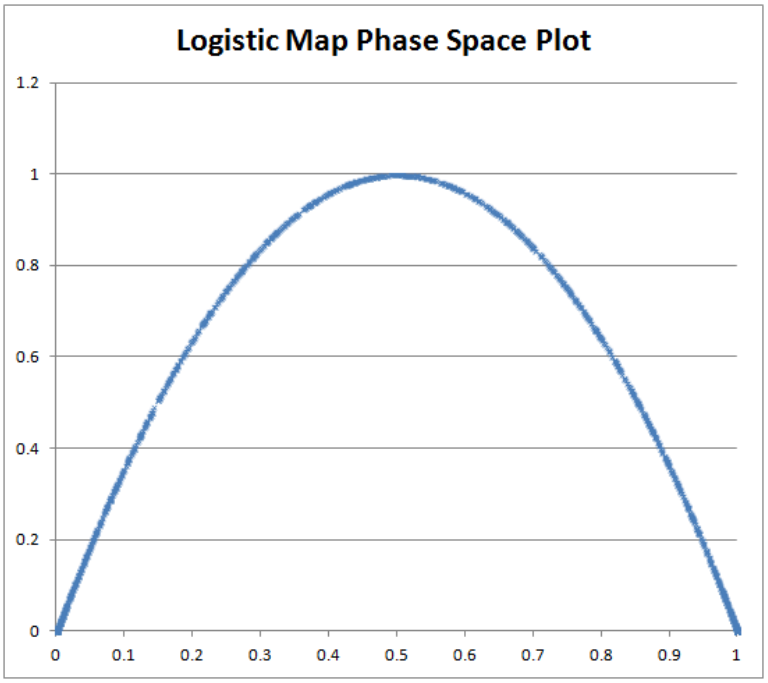

Figure 2.6: The phase space diagram for the chaotic logistic map - showing the relationship between X_t and X_{t-1}.

2.5.3 Lorenz System

Lorenz found three ordinary differential equations which closely approximate a model for thermal convection [15]. These equations have also become a popular benchmark for testing non-linear predictors. The Lorenz model is given by the equations in 2.10.

$$\begin{aligned} \frac{dx(t)}{dt} &= a(y-x), \\ \frac{dy(t)}{dt} &= bx - y - xz, \\ \frac{dz(t)}{dt} &= xy - cz \end{aligned} \tag{2.10}$$

For this system, we will generate two sets of data in order to make comparisons with the results obtained in [12] and [16]. One series will be generated with $a = 10$, $b = 28$ and $c = 8/3$. Another will be generated with $a = 16$, $b = 45.92$ and $c = 4$. In both cases, a time step of $\Delta t = 0.01$ will be used. These sets of parameters are commonly used in generating the Lorenz sys-

tem because the resulting systems have been shown to never form closed cycles or reach a steady state; instead, these systems exhibit deterministic chaos.

2.5.4 Henon Map

The Henon chaotic time series can be constructed with the following equations:

$$\begin{aligned} x(t+1) &= 1 - ax(t)^2 + y(t), \\ y(t+1) &= bx(t) \end{aligned} \tag{2.11}$$

When generating data for our experiments, we set $a = 1.4$ and $b = 0.3$. These same parameters are used in both [12] and [17].

2.5.5 Sunspot Number

The sunspot number is a naturally occurring chaotic series. It provides a measure of solar activity and has a period of 11 years - known as a solar cycle. As solar activity has a significant affect on the climate, satellites and space missions, it is very useful to be able to forecast the series accurately. While several sunspot number data series exist, the most useful series to predict is the smoothed monthly sunspot number. This series has been shown to exhibit deterministic non-linear chaotic behaviour [18] and so is inherently very difficult to predict.

2.5.6 River Flow Rates

Another interesting and naturally occuring series is that of daily river flow rates. River flow prediction is important as it is used to forecast and mitigate floods, and in the planning and operation of water provision projects. In our analysis, we use a time series representing the Mississippi river daily flow rate. The Mississippi river is one of the world's largest rivers with a length of over 3700km and has become a popular subject of flow rate prediction research. It has been shown in previous studies that its daily flow rate demonstrates chaotic behaviour [19][20].

2.5.7 Attempts at Chaotic Prediction

The evolution of statistics, artificial intelligence and machine learning has given rise to numerous attempts to predict chaotic time series. We will later evaluate the novel technique by comparing its performance with that of previous efforts.

A statistical approach to modelling non-linear time series is presented by J. Farmer and J. Sidorowich [21]. They demonstrate their technique by applying it to several examples, including data from the Mackey-Glass system.

It has been shown that fuzzy inference systems based on simple 'If-Then' rules can achieve a higher predictive accuracy than the traditional statistical approaches. Jang and Sun show how this technique can outperform auto-regressive modelling. Similarly to the above, they use the Mackey-Glass system as a basis for comparison.

Gholipour *et al.* demonstrate how fuzzy neural networks (an adaptation of the MLP) can achieve a high predictive accuracy on the Mackey-Glass system, the Lorenz system and also on the sunspot number time series [12]. As can be seen in Figure 2.7, their best neurofuzzy model is able to perform a multi-step prediction to predict an entire solar cycle with very high accuracy. This is a significant improvement on previous attempts to predict the sunspot number series, such as the statistical approaches presented in [18] and [22]. The neurofuzzy models were also shown to be able to predict the Lorenz system accurately in the presence of artificial noise.

A different method was adopted by Karunasinghe and Liong, who demonstrated that a global artificial neural network model is often able to outperform the widely used local prediction models [16]. They present their findings on the Lorenz series (both with and without noise) and on two chaotic river flow time series - one of which is the daily flow rate for the Mississippi river.

Another interesting approach, presented by Aly and Leung [23], involved the fusion of results from various predictors (including neural networks and statistical techniques) to produce a final predictor which is more accurate than any of the individual predictions. The fusion approach was demonstrated on the Mackey-Glass series. The paper concludes that the fusion of predictors is more robust when there is insufficient training data available

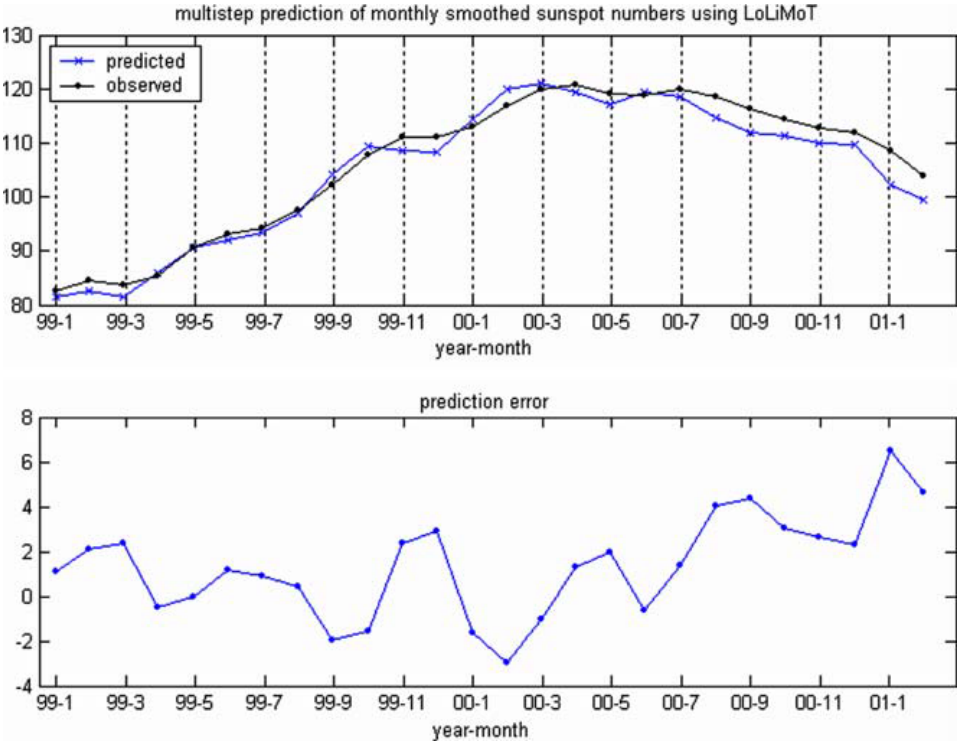

Figure 2.7: Multi-step prediction of the monthly smoothed sunspot numbers for the 23rd solar cycle, obtained using a fuzzy neural network.

than any of the individual predictors.

There also exists techniques inspired by evolutionary science and genetics. Cortez *et al.* have shown how artificial neural networks can be combined with genetic algorithms [17]. With this technique, an algorithm is used to find the best neural network solution to a prediction problem by tweaking parameters and evolving a population of potential solutions towards the best solution using a 'fitness' function. In the case of time series prediction, the fitness function is higher when the RMSE is lower and when the model is smaller (to avoid unnecessary complexity). In [17], the evolutionary neural network approach is compared with ARIMA using several real-world and chaotic series, including the Henon map.

2.6 Weka

The Weka project provides a comprehensive collection of machine learning algorithms and enables researchers to quickly try out and compare a variety of machine learning methods on a data set [24]. Weka has a time series framework which applies the machine learning/data mining approach to time series forecasting, as discussed in section 2.2.5. The project is open source and is written in the Java language; it has been well developed by the data mining community and performs many of the same features as expensive software used by large companies. In addition, Weka provides an API enabling it to be easily embedded into other applications.

3 Design

Our new implementation is based on the original prediction algorithm, for which the pseudocode was provided by Ben Rogers. Initially, our aim was to create an object-oriented version of the algorithm which solves two key limitations of the BASIC implementation. The original algorithm is only able to work with fixed 8-bit integers and only supports single-step prediction. On the other hand, our new implementation supports an arbitrary bit size for observations and predictions and also provides the functionality to perform multi-step prediction. In this chapter, the core design and key aspects of the implementation are discussed. Later, additional features including model visualisation and several analytical tools and extensions are developed.

3.1 Tools & Design Decisions

In this section, we discuss our choice of programming language along with which tools and libraries can be used in our implementation.

3.1.1 Programming Language

With an initial objective of developing an object-oriented implementation of ALPE, one of the first decisions to make was which language to use for the new implementation. The obvious choices were C#, Java or C++, all of which are powerful and widely used object-oriented languages.

One of the most significant differences between these languages is in their compilation procedure. At compile time, both Java and C# compile to an intermediate language which is independent of the target architecture and operating system and executes in a managed environment. This can result in very fast compilation, but performance at run-time can suffer as the intermediate language has to be translated into a set of machine code instructions. On the other hand, C++ compiles and links the source files

directly into a native executable and it is possible to compile for both Windows and Linux based platforms. Compilation is generally slower than Java and C# but the compiler has more time to make optimisations. C# and Java provide garbage collection for the convenience of the developer, this adds a performance overhead as objects and references must be tracked by an additional process. C++ does not have this overhead, again resulting in improved performance with the trade-off being that the developer has an additional responsibility to manage memory allocation carefully. In this project, performance is very important; we want our implementation to handle large predictive models while maintaining a speedy predictor. Therefore, C++ is a suitable choice for our programming language.

Another advantage of using C++, particularly for this project, arises from the fact that implementing ALPE requires bit-level manipulation. C++ provides the developer with significantly more control over low level bit manipulation than Java and C#.

Finally, C++ has been around for a much longer period of time and subsequently a very large collection of libraries has been made available for developers to use. This means there is a great choice of graphing libraries and support for statistical analysis, which are also aspects we will need to consider.

3.1.2 Graphical User Interface

There are two popular toolkits for C++ graphical user interface development: the Qt Framework and WxWidgets. Both are cross-platform compatible and having been around for over twenty years are very mature and widely used. Both have many useful utilities which enable developers to create imaginative and stylish user interfaces. However, their biggest difference is in the tools and documentation packaged with their distributions.

Unlike WxWidgets, Qt is distributed with an integrated development environment called Qt Creator - which facilitates the design and creation of interfaces making development considerably faster and more convenient for the developer. Qt is also heavily documented and there is an extensive amount of support material available, whereas the documentation for WxWidgets is less extensive. With no experience in developing an interface for a C++ application, Qt was the preferable choice.

3.1.3 Additional Libraries

In addition to C++ and Qt, some additional libraries are required for plotting graphs, visualising the predictive models and performing statistical analysis.

We will use QCustomPlot, a plotting widget which is designed to be used with Qt. It has a simple and easy to use API, and provides sufficient graphing functionality for the plotting of observation, prediction and error graphs in our implementation.

To visualise the predictive models, we will use UbiGraph - a tool for the dynamic visualisation of graphs. UbiGraph runs in its own process, but provides a C API which enables communication with the UbiGraph process over the XML-RPC protocol. We can then make calls to the API to add nodes and edges to the graph whilst the algorithm is running - resulting in a real-time visualisation of the model.

The free, open-source R programming language and environment provides a huge range of pre-packaged statistical functions and data analysis tools. It is widely used by statisticians and data miners for data analytics and its library of functions includes the auto-correlation and partial auto-correlation functions which we will need to analyse our predictions and models.

In order make calls to and communicate with the R environment from within our C++ implementation, we use the RInside and Rcpp packages. These packages enable the embedding of R within C++ code. It is also possible to use RInside with the Qt framework to display graphs generated by R within our own graphical user interface.

3.2 Class Overview

A simplified UML diagram showing the core classes and relationships in our new implementation is shown in Figure 3.1. A more complete class diagram (depicting all classes in the final implementation, including those added as part of the extensions discussed in Section 4.4) can be seen in Figure A1.

The `Predictor` class provides direct access to the prediction algorithm through a simple API which does not require an understanding of the algorithm itself. The API is discussed further in Section 3.3. The `Predictor` is composed of n `Tree` instances, where n is a parameter specifying the

number of bits a given `Predictor` instance can handle.

The `Predictor` also requires a reference to an abstract class, `TimeSeriesProvider`, which guarantees provision of a function which can be used to retrieve a series of observations. In our initial implementation, there is just one implementation of this class: `TimeSeriesFromFile`.

Any developer should be able to create an instance of and use `Predictor` with no additional configuration. In our implementation, we chose to use the Qt framework to design a user interface which interacts with an instance of `Predictor` via a separate thread. To accomplish this, a `PredictorInterface` class is introduced. This class has the responsibility of handling all logic involving the user interface as well as coordinating communication between components via the Qt signals and slots mechanism. In order to run the prediction algorithm, an instance of `PredictorThread` is created and started by the `PredictorInterface`. Predictions can then be made asynchronously, and the thread can emit a signal to notify the user interface upon each new prediction.

3.3 API Functions

To keep the API simple, just a few key functions are made available by the `Predictor` class. These are listed and explained in Table 3.3.

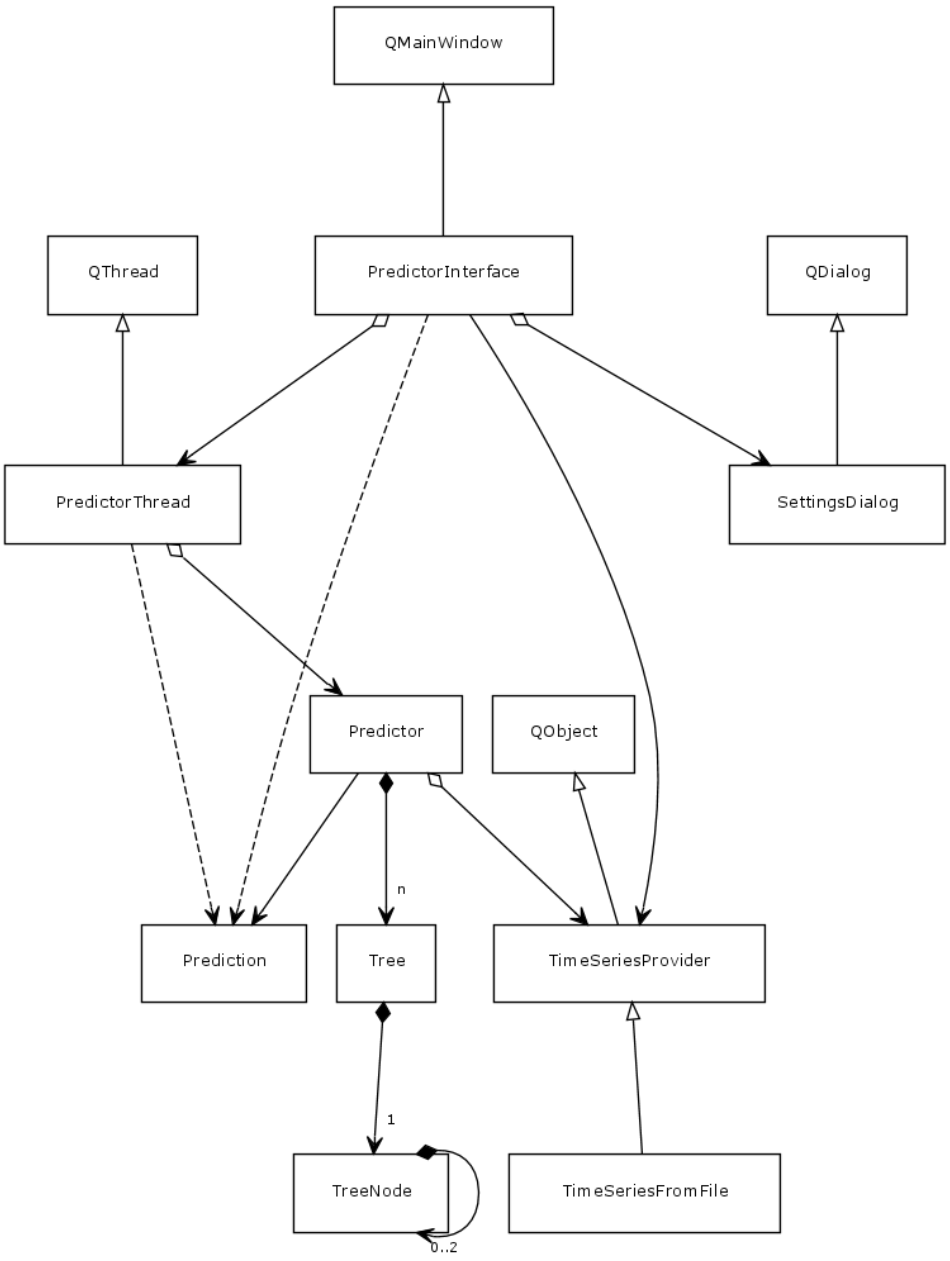

Figure 3.1: UML diagram showing the relationships between classes in the basic C++ implementation.

Method Signature	Description
`Predictor(TimeSeriesProvider&, int)`	Constructor. Creates new instance of `Predictor` and sets its time series provider accordingly. The `int` parameter specifies the expected bit size of each observation.
`Prediction makePredictionAndLearn()`	Triggers the predictor to make a new prediction, given the observations made so far. The predictor will then make a corresponding observation, compare this with its prediction, and modify the predictive model appropriately. The prediction is returned in a `Prediction` object.
`void reset()`	Restores the predictor instance and the current model to its original state.
`void setMaxNodeCount(int)`	Sets the maximum size of the predictive model. (Default value is $2,000,000 * n$ where n is the bit length of observations.)
`void setSignedPredictor(bool)`	Can be used to toggle the predictor between signed and unsigned mode. In signed mode, negative observations are allowed. (Default value is unsigned, i.e. false.)
`void setPlotUbigraph(bool)`	Enables or disables the model visualisation. Faster performance is expected with visualisation disabled. (Default value is true.)
`int getBitCount()`	Returns the observation bit size which the predictor expects.
`int getMaxNodeCount()`	Returns the current limitation on the size of the predictive model.
`Prediction makePrediction(vector<long>, bool)` **(extension)**	Triggers the predictor to make a prediction using the current model, but pretending that the last observations made are those given in the vector parameter. The boolean parameter specifies whether these observations should be temporarily appended to the actual observations. The state of the predictor and model are not affected by a call to this function.

Table 3.1: Description of the key API functions.

4 Implementation

Development of the C++ implementation was split into several stages. Firstly, a prototype implementation of a single-bit predictor was developed. This is a predictor that is only able to learn and model a binary time series. This implementation was then scaled to support observations and predictions of an arbitrary bit size. Following this, additional features including model visualisation, artificial noise generation and tools for analysis and experimentation were developed. In this chapter, we discuss details of the implementation and provide an overview of the features and extensions.

4.1 Single Bit Predictor

The first stage of development involved creating a prototype single-bit predictor. The core classes were implemented as discussed in Section 3.2.

Figure 4.1 shows the single bit predictor working as expected on a simple repeating binary series. QCustomPlot is used to draw observation, prediction and error graphs whilst the predictor is running. Here, the red area in the error graph indicates that the prediction was different to the observation. As expected, the predictor learns the repeating series very quickly.

Once satisfied that the single bit predictor was working as expected, the next challenge was to scale this to work with an arbitrary number of bits.

4.2 Arbitrary Bit Predictor

In scaling the predictor to work with an arbitrary number of bits, it was important to keep the decrease in performance for larger bit capability as small as possible.

One of the challenges in ensuring scalability was in implementing a node recycling scheme. This is required in order to limit the size of the model while at the same time ensuring that the model can still be improved with

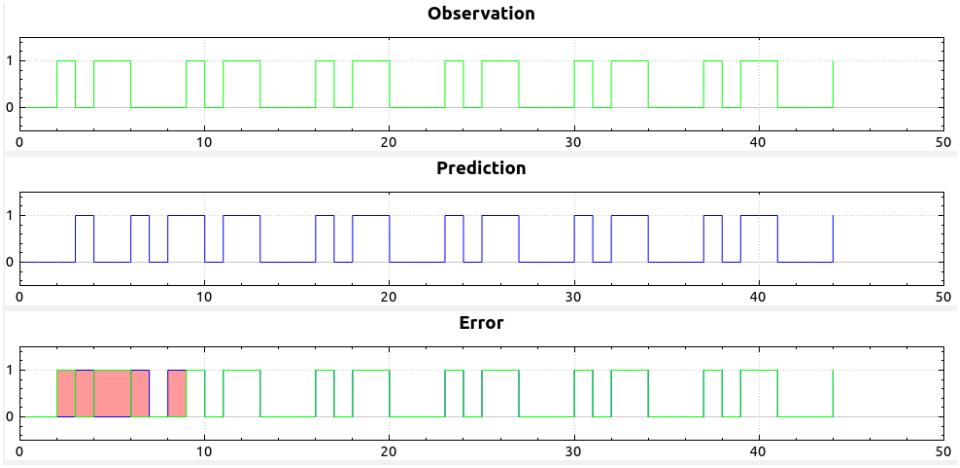

Figure 4.1: The single bit predictor being tested on a repeating binary series of eight observations: 00101101

new observations. The recycling of a node involves deleting a leaf node and replacing it with a new node. Our initial solution involved maintaining a list of leaf nodes and then picking a random node for deletion from this list. However, maintaining the list of leaf nodes proved to be very costly and the predictor operated slowly for a large number of bits. A better solution is to maintain a *pool* of all nodes in the model, and select randomly from this pool until a leaf node is found. As there is a high probability of selecting a leaf node, this technique works very effectively and scales well.

Another concern was with managing the bits effectively in memory. In the single bit implementation, the `std::bitset` is used to hold observations and predictions in memory. This is a specialist container class which makes bit-level manipulation easier to program and more efficient in terms of space optimisation and performance. However, at compile time, this type requires a template parameter for its size (e.g. `std::bitset<8>` declares a bitset with a capacity of 8 bits). In the arbitrary bit predictor, we do not know the number of bits we need to work with at compile time and so the `std::bitset` cannot be used. The popular C++ boost library provides us with a solution: the `boost::dynamic_bitset<>`, a container similar to the standard bitset but which allows dynamic specifiction and resizing of the container size. This type is therefore used for all bit-level manipulation in the multi-bit predictor.

To organise the application settings, a new `QDialog` was created. This provides a convenient interface which can be used to change settings such as the number of bits the predictor should handle and the model size limit. A screenshot of the final settings dialog is shown in Figure A2.

Figure 4.2 shows the multi-bit predictor being tested on a sine wave signal. It was decided to replace the error plot from the prototype with a more typical residual plot. To aid analysis, some additional statistics were added to the user interface, displaying the size of the model and the RMSE error metric.

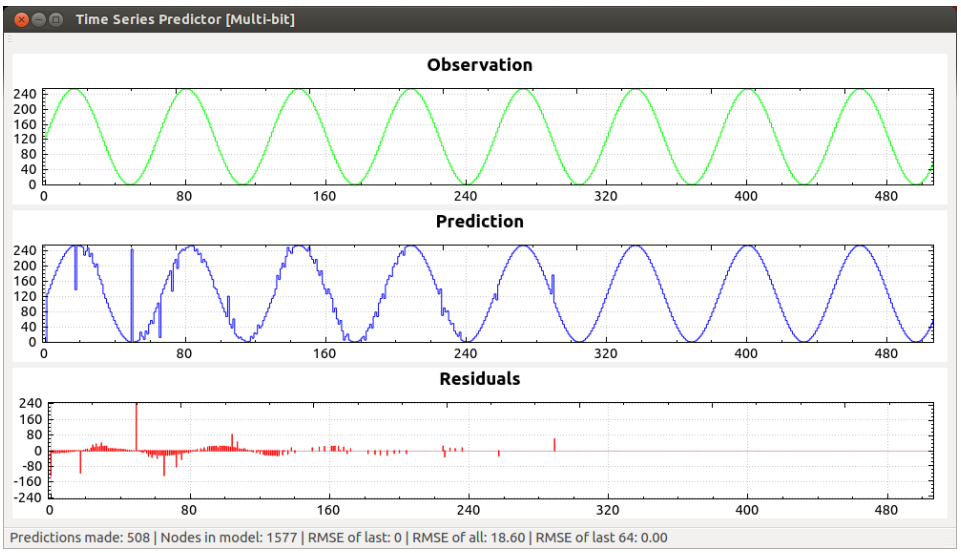

Figure 4.2: The multi-bit predictor being tested on a sine wave time series.

4.3 Model Visualisation

As discussed in Section 3.1.3, UbiGraph is used for 3D visualisation of the predictive models. Upon initialisation of the C++ application, an instance of Ubigraph is started. Whenever the model is updated or modified by the predictor, the C++ application communicates with the Ubigraph instance using the Ubigraph API. For example, a call to `ubigraph_new_vertex()` is used to create a new node in the visualisation, and a call to `ubigraph_new_edge(vertex_id_t x, vertex_id_t y)` is used to link two nodes together. Communication with the Ubigraph

instance takes place over the XML-RPC protocol.

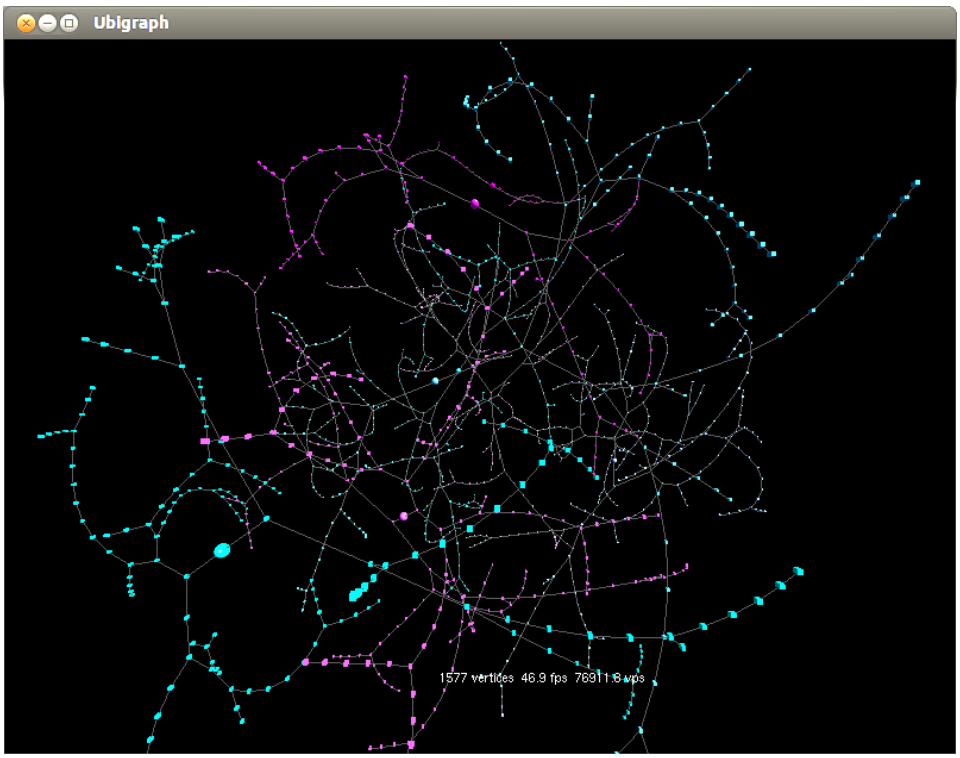

Figure 4.3: Visualisation of the model constructed for the sine wave series.

4.4 Additional Features

To test that the new implementation implements the ALPE approach correctly, we compared the output of the C++ predictor with the output of the original BASIC implementation for a range of 8-bit series. Having observed a perfect match between both sets of output across several test cases, we can be confident that the new implementation is performing correctly. We were then able to proceed and extend the implementation further.

4.4.1 Negative Numbers

An immediate limitation of the basic implementation is that negative observations and predictions are not supported. In order to handle negative numbers, we use the `signed long` type which follows the standard two's com-

plement representation. However, there is a complication when converting a bitset representation to a signed long, since the `boost::dynamic_bitset` only provides the functionality to convert to an *unsigned* long. A static cast alone is not enough to solve this problem since the number of bits used to represent our values is usually less than the number of bits in a C++ `long`. This causes the cast to work incorrectly when dealing with negative numbers. For example, an 8-bit signed long representation of -5 is not equivalent to a 64-bit signed long representation of -5. To solve this problem, we must perform a sign extension operation before performing a type cast. By doing this, the sign is preserved for negative numbers.

A conversion of bitset 'bs' to a signed long is then given by:

```
long mask = (long)(-1) << bs.size();
static_cast<signed long>(
  bs.to_ulong() | (bs.test(bs.size() - 1) ? mask : 0)
);
```

Also, the acceptable range of observations and predictions must be changed when switching between a signed and unsigned predictor. For an n-bit unsigned predictor, the range is $[0, 2^n - 1]$ whereas for an n-bit signed predictor, the range becomes $[-2^{n-1}, 2^{n-1} - 1]$.

4.4.2 Noise Generation

The ability to add artificial noise to a clean signal before it reaches the predictor is very useful for analysis. In our implementation, we introduce a new abstract class, `NoiseGenerator`, which provides a virtual function `long addNoise(long)`. We then alter the implementation of `TimeSeriesProvider` so that it can store a reference to a `NoiseGenerator` instance. If a noise generator is set, then its addNoise function is called for each noise-free observation before the observation is returned to the predictor. This design is depicted in Figure A1.

For now, there is just one subclass of `NoiseGenerator`, the `UniformNoiseGenerator`. This subclass can be used to add uniformly distributed white noise to observations. To do this, the `boost::random::variate_generator` is used to combine a `boost::uniform_int<>` distribution function with a random number generator. For high quality random number generation an implementation

of the Mersenne Twister (discussed in Section 2.3), `boost::mt19937`, is applied. Of course, if another noise distribution is required, we can easily define a new subclass of `NoiseGenerator`. If we apply uniformly distributed noise in the range of $[-5, 5]$ to a flat (zero) signal, then as expected, our residuals are completely random over time (see Figure 4.4).

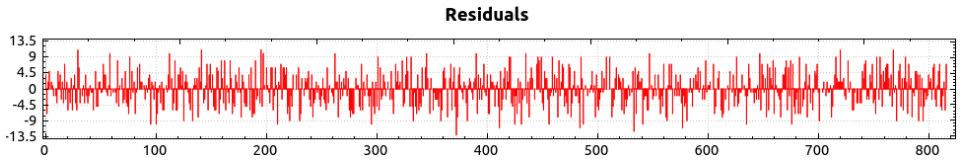

Figure 4.4: Residual graph resulting from applying the predictor to a uniform white noise signal.

4.4.3 Statistical Analysis

As discussed in Section 2.1.3, plotting the ACF and PACF of residuals is a statistical approach to evaluating our predictive models. We therefore extend our implementation to incorporate this functionality.

The R environment is first embedded into our C++ application, using the method described in Section 3.1.3. A series of residuals can be loaded into the R environment and R's ACF and PACF functions applied to the series. The resulting plots are saved to a temporary SVG file and can be displayed in the C++ application using a `QSvgWidget`.

4.4.4 Multi-Step Prediction

Many relevant research papers present results for multi-step predictions rather than for single-step predictions [12][16][23]. The ability to make multi-step predictions is evidently very desirable in time series prediction, particularly when forecasting real-world time series. We therefore adapt our implementation of ALPE to support multi-step prediction - with the aim of enabling the algorithm to be used to make predictions for an arbitrary number of steps in the future at any point in time.

Since the algorithm is, by design, an online, single-step learner, an obvious method of making multi-step predictions is to feed single-step predictions

back into the predictor as observations. For example, if we make 10 observations and wish to predict the next 2, this can be achieved by predicting the 11th observation and then using this prediction along with the 10 observations to predict the 12th. By doing this iteratively, an arbitrary number of future observations can be predicted.

To implement this, the predictor's API has been extended with an additional function: `Prediction makePrediction(vector<long>, bool)`. As described in Table 3.3, the vector parameter allows us to feed predictions back into the predictor to use in the next prediction. By adding each new prediction to this vector, we are able to use the current model at any point in the series to make a multi-step prediction.

4.4.5 Phase Space Plot

As discussed in Section 2.5, a phase space plot can be used to give a quick assessment of how well the predictor is performing on chaotic series. To implement this, an instance of `QCustomPlot` is added to a new `QDialog`. The `PredictorInterface` creates a single instance of this dialog upon instantiation and is able to update the phase space plot in real-time upon receiving each prediction.

4.4.6 Compression Analysis Utility

A final extension to the C++ implementation is the addition of a compression analysis utility. For purely experimental purposes, we want to investigate whether the models generated with ALPE could be useful in the domain of compression. Our motivation for this is discussed further in Section 5.7. One way of exploring this idea is to apply the algorithm to an ASCII string, by passing the string to the predictor (as a series of ASCII characters) one or more times. Having constructed a model, we want to see how well the algorithm can reconstruct the string, given just the first few characters. We also need to compare the amount of memory needed to store the model relative to the amount needed to store the string. To accomplish this, a utility is developed which can perform all of these functions.

Before developing the utility, a way of using an ASCII text file as a time series had to be established. This involved creating a new class, `TimeSeriesFromASCII` - a subclass of `TimeSeriesProvider`. This

class simply interprets each character (byte) in the file as an 8-bit observation. The utility therefore requires an instance of `Predictor` with an 8-bit capacity. Similarly, a `TimeSeriesFromBitStream` class was implemented which enables us to treat an ASCII file (or a binary file) as a sequence of bits. In doing this, we are able to later experiment with each method and see which yields the best results for compression. A new dialog, `ASCIICompressionDialog` is used to provide a user interface for the utility. The dialog enables us to specify the number of times a string should be passed through the predictor, along with the number of characters to store for initiating decompression. In the decompression step, the same approach described in Section 4.4.4 is used to make multiple 'predictions' given just a few initial characters. These predictions correspond to our decompressed characters and are used to reconstruct a string.

The C++ algorithm for this procedure is shown in Figure 4.5. This algorithm demonstrates how the predictor's API can be used for multi-step prediction.

When running the compression analysis procedure, the compression accuracy and model size are calculated each time the string is passed through the predictor. At termination, the most accurate model is displayed along with the number of nodes in the model. A screenshot of the utility in action is shown in Figure A3.

```
unsigned int node_count = 0;

// For each 'run' (i.e. passing of the string through the predictor)
for (int run = 0; run < spinRuns->value(); run++)
{
    // Generate model by running entire series through predictor
    for (int i = 0; i < series.size(); i++)
        predictor->makePredictionAndLearn();

    // Use initial characters to generate a history
    std::vector<long> history(spinCharacters->value() * BITS_PER_CHAR);
    for (int i = 0; i < spinCharacters->value() * BITS_PER_CHAR; i++)
        history[i] = series[i];

    // Combine model with 'initial characters' to decompress the file
    int predictionsRequired = series.size() - history.size();
    for (int step = 0; step < predictionsRequired; step++)
    {
        Prediction prediction = predictor->makePrediction(history, false);
        history.push_back(prediction.getPrediction());
        node_count = prediction.getNodeCount();
    }

    // Find 'decompressed' ASCII string
    std::string result;
    for (int c = 0; c < history.size(); c += BITS_PER_CHAR)
        result.push_back(getChar(history, c));

    // Calculate decompression accuracy and model complexity
    ...
}
```

Figure 4.5: C++ algorithm used in compression analysis.

5 Experimentation

Given that the true potential of ALPE is completely unknown, we conduct numerous experiments in order to explore the applicability and power of the algorithm with the aim of discovering its strengths and weaknesses. We begin by investigating which types of data the algorithm can handle and under what conditions it performs best. We add artificial noise to various time series and observe the effects this has on the algorithm's model and predictions. A statistical analysis of the models constructed by the algorithm is then carried out. To gain an understanding of how ALPE performs relative to other methods for time series prediction, we perform a thorough comparison with established and emerging techniques.

5.1 Data Generation & Preprocessing

Prior to beginning the experimental work and analysis of the predictor, several test data sets were generated and preprocessed. In some cases, having generated a series, the data points must be scaled to make them compatible with the predictor. This is because a generated series is often a set of continuous floating point numbers whereas we require a set of discrete integers. MATLAB was used for the generation of chaotic time series and Excel was used for the preprocessing of all data sets. In the generation of chaotic time series, we use the parameters specified in Section 2.5.

For each series, having generated the data the best way to scale the series (i.e. the number of bits to use to represent the series) must be determined. If too few bits are used, then the scaled series will be significantly less accurate than the original series and our results will not be reliable. If too many bits are used, then the model constructed will be much larger than it needs to be. The procedure of determining the optimal scaling is demonstrated here for the Mackey-Glass series.

Having generated the Mackey-Glass series, we have a set of data points

in the interval of $[0.21922, 1.313768]$. This series can be scaled and rounded so that it is compatible with the predictor. This means that if we scale the series to lie in the range of $[0, 2^{16} - 1]$ then this will yield a series which is compatible with a 16-bit predictor. To evaluate a particular scaling, we pass 2100 data points through the predictor of which the last 100 of these are used to calculate an error metric. Having calculated the NRMSE for a number of different scalings, the results shown in Figure 5.1 are obtained. We observe that in this case, the NRMSE stabilises for approximately 20 or more bits. Therefore, for this series we use a 20-bit predictor to carry out our analysis.

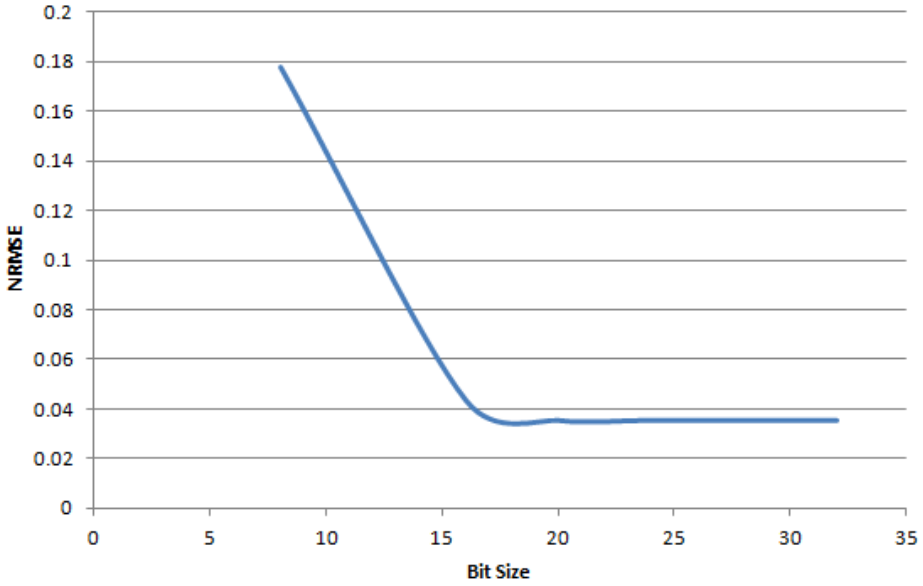

Figure 5.1: The variation of the NRMSE with respect to different scalings of a Mackey-Glass time series.

5.2 Model Analysis

A general analysis of the models constructed by the algorithm is performed. We analyse the performance of the models in terms of predictive accuracy, their complexity and the number of observations required to find structure in chaotic behaviour.

5.2.1 Correctness

As described in Section 2.1.3, the autocorrelation of residuals give us a statistical indication of how well the learned model is capturing structure in the data. By passing 2000 observations through the predictor and then plotting the ACF and PACF for the last 100 residuals we can see if there is any significant correlation at any lag. Having done this for several simple and chaotic series, such as the logistic map as shown in Figures 5.2 and 5.3, we observed that for all series there is generally no significant correlation for any lag other than the zero lag.

The confidence bounds shown in the plots are at the 95% significance level. In some cases there is one statistically significant correlation (such as the 4th lag for the logistic map) but such anomalies are expected since 20 lags are being considered at the 95% level. As there is no significant evidence for the autocorrelation of residuals, this implies that the models being created by the algorithm are correct in the sense that they are capturing the structure of the time series well.

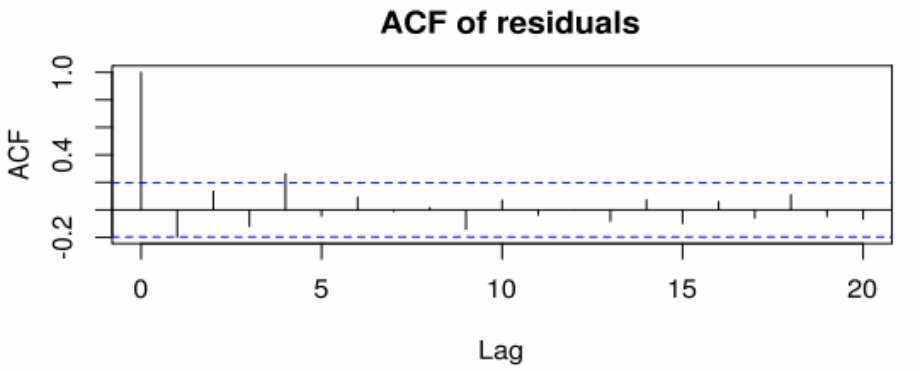

Figure 5.2: Plot of the ACF of residuals for the chaotic logistic map.

5.2.2 Complexity

We can use our visualisation of the model to keep track of the number of nodes (vertices) in use at any point in time. This corresponds to the size of the model. Table 5.1 shows how the size of the model varies for series of different complexities. As we expected, more complex problems require more nodes in the model. For simple series, such as the sawtooth and sine

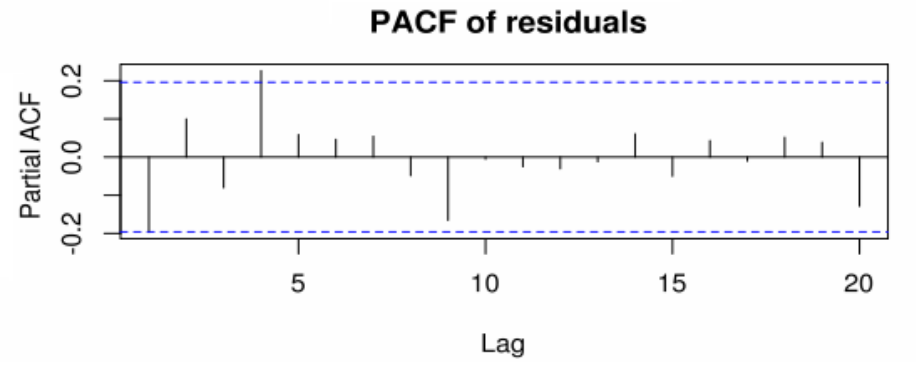

Figure 5.3: Plot of the PACF of residuals for the chaotic logistic map.

waves, the algorithm is able to construct a perfect model and so the size shown is exact. For chaotic time series, it is impossible for the algorithm to construct a perfect model and so we can only show an approximation of the model size. This corresponds to the approximate size of the model when the RMSE becomes stable.

	Bit Size	**Node Count**
Sawtooth Wave	8	864
Sine Wave	8	1577
Logistic Map	16	~ 37000
Mackey Glass	20	~ 200000

Table 5.1: Variation in model size with series of different complexities.

During experimentation we also observed that in the presence of noise (of any level) the model size grows infinitely large, until the node limit is reached. This is because it is in the nature of the algorithm to continually build the model while predictions are not perfect. Therefore, in some cases, it is important to set an appropriate node limit to prevent the algorithm from building an unnecessarily large model. The effect of noise is further discussed in Section 5.4.

5.2.3 Phase Space

Through examining the phase space plot during the prediction of chaotic series, we can get a good idea of how quickly the predictor is learning the deterministic behaviour in the chaos. For the chaotic logistic map, it is difficult to gain an understanding of how well the predictor is performing by examining the observations and predictions alone. As shown by Figure 5.4, the series appears to be extremely random. If we instead examine how the predictor's phase space plot changes over time, we are able to see the distinctive parabola formation after just 200 observations (Figure 5.5). This demonstrates how quickly the algorithm is able to learn the structure.

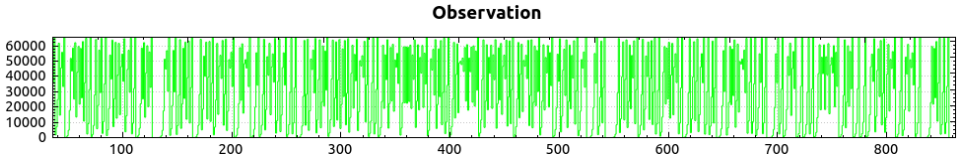

Figure 5.4: 1000 observations of the chaotic logistic map time series.

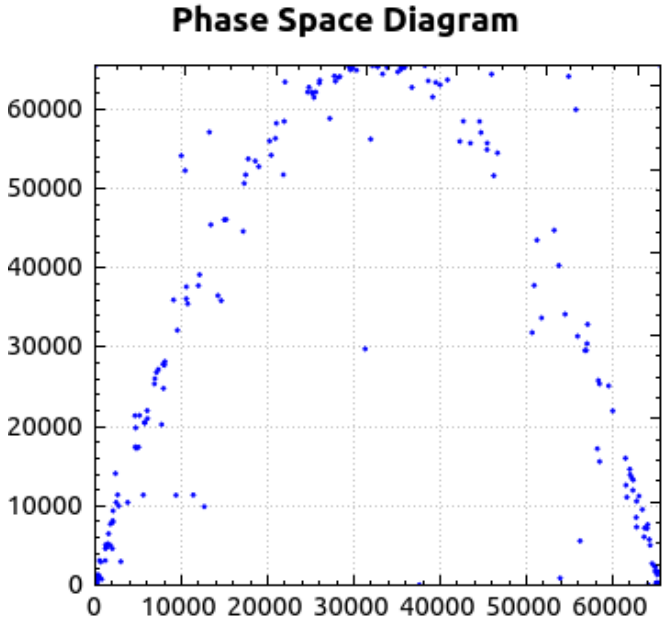

Figure 5.5: The predictor's phase space plot after 200 observations of the chaotic logistic map.

Similarly, for the Henon map we see that after 1000 observations a clear resemblance of the expected phase space has become visible (Figure 5.6). Again, this highlights the algorithm's ability to quickly find structure in chaos.

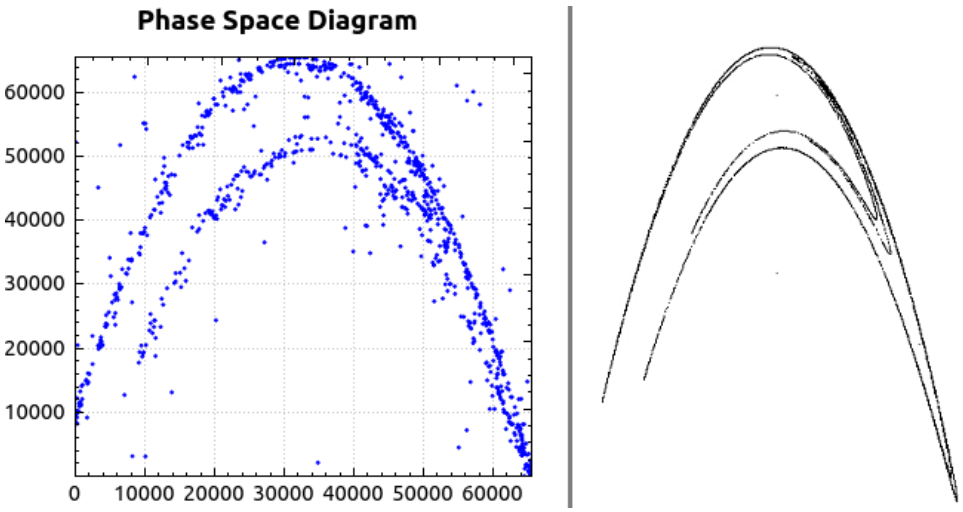

Figure 5.6: Left - The predictor's phase space plot after 1000 observations of the Henon map. Right - The expected appearance of the Henon map's phase space [25].

5.3 Comparison with Weka

The predictor's accuracy is compared with the data mining methods discussed in Section 2.2.5 using Weka's time series framework. A comparison is made with the MLP and SVM approaches for both single-step and multi-step predictions. In each test, the first 2000 observations were used for training and then the NRMSE error metric was calculated over the next 20 data points. As an exception, for the multi-step Mackey-Glass test 200 data points were used in our calculation. This is because the Mackey-Glass series follows a smooth curve and so more time is needed to observe its chaotic behaviour. Since the MLP and SVM give continuous predictions, these predictions were rounded to integers before calculating the NRMSE, so that we are able to make a fair comparison with ALPE.

Table 5.2 presents our results for the noise-free sine wave (8-bit), chaotic logistic map (16-bit) and Mackey-Glass series (20-bit) for both single-step and multi-step prediction.

	Single-Step			Multi-Step		
	ALPE	**MLP**	**SVM**	**ALPE**	**MLP**	**SVM**
Sine Wave	0.0000	0.0082	0.0046	0.0000	0.0243	0.0137
Logistic Map	0.0374	0.0410	5.3592	0.7679	1.2144	10.7654
Mackey-Glass	0.0346	0.0107	0.0127	1.4341	1.0394	0.9998

Table 5.2: The NRMSE for ALPE, the Multi-Layer Perceptron (MLP) and Support Vector Machine (SVM).

For the sine wave all three predictors achieve an NRMSE very close to zero, which is what we would expect given that this is a simple cyclic noise-free series. Here, ALPE has an NRMSE of zero for both single-step and multi-step prediction as it is able to model the discrete nature of the time series perfectly, whereas since the MLP and SVM are intended for continuous data there is often a small error in their predictions.

For the logistic map, ALPE and the MLP are able to make single-step predictions very well, showing that they are both able to model the determinism in the chaotic series. With multi-step prediction, ALPE is the only technique which scores an NRMSE less than one, meaning that it is more accurate than using the mean of the series for every prediction. However, it is still a high NRMSE which indicates that the algorithm is not able to make multi-step predictions for the logistic map very well. This can be explained by the short-term predictability of chaotic series.

Finally, for the Mackey-Glass series, we see that the MLP and SVM achieve a slightly lower NRMSE for single-step prediction than ALPE, however clearly all three techniques are able to make very accurate short-term predictions for this series. In multi-step prediction, we see that all three techniques are unable to make accurate long-term predictions.

5.4 Adding Artificial Noise

To find out how well the predictor handles noise, uniformly distributed white noise in the range of $[-x, x]$ is added to each series, where x is 10% of the standard deviation of the clean series. The same tests as described in Section 5.3 are then performed on noisy versions of the sine wave, logistic map and Mackey-Glass series.

An initial observation for the sine wave experiment is that the predictions are consistently more noisy than the observations. We can also see that there appears to be a trend in the residuals. To confirm this, we examine the autocorrelation of residuals (see Figure 5.8) and find that there is significant autocorrelation in the residuals at several lags. As the residuals themselves have structure (as opposed to being uncorrelated white noise), this implies that the predictor is not finding the best possible model for the noisy sine wave. For the logistic map and Mackey-Glass series, no significant trend in the residuals was found.

The results for all three series are presented in Table 5.3. We see that for single-step prediction, all three series have an NRMSE of approximately 0.2. This shows that the models are still reasonably good and that the predictor is still able to capture the deterministic behaviour of the chaotic series in the presence of noise.

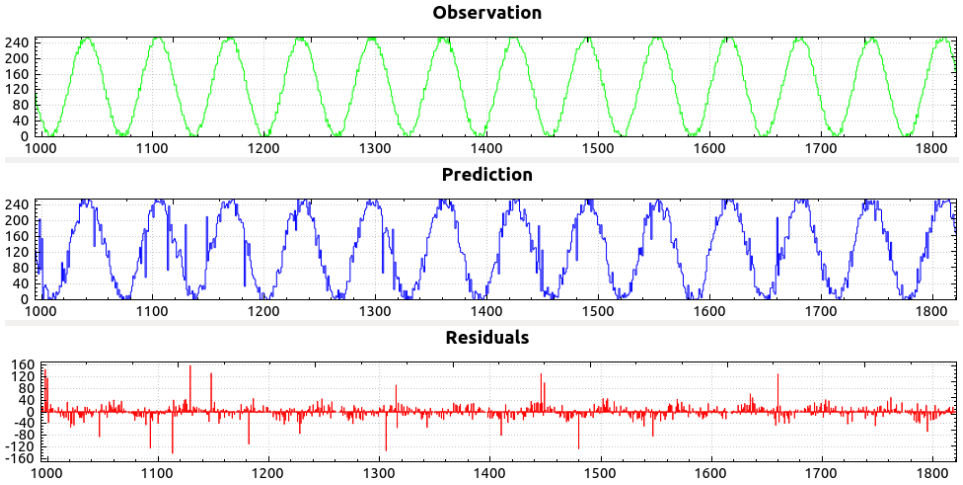

Figure 5.7: Plot of observations, predictions and residuals on a noisy sine wave.

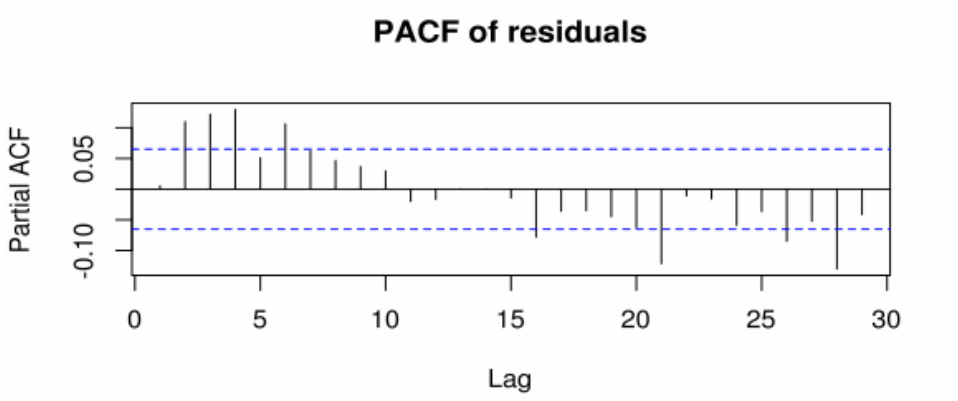

Figure 5.8: The PACF of residuals when predicting a noisy sine wave.

	Clean		Noisy	
	Single-Step	**Multi-Step**	**Single-Step**	**Multi-Step**
Sine Wave	0.0000	0.0000	0.2280	0.0901
Logistic Map	0.0374	0.7679	0.2272	1.1963
Mackey-Glass	0.0346	1.4341	0.2185	1.6209

Table 5.3: A comparison of the NRMSE on clean and noisy time series using ALPE.

5.5 Non-Stationary Series

We investigate how well the predictor is able to forecast non-stationary time series. Here, we apply the predictor to an airline passenger dataset which consists of 144 monthly passenger numbers for an airline for the years of 1949 to 1960. As can be seen in Figure 5.9, this is a non-stationary time series with a linear trend.

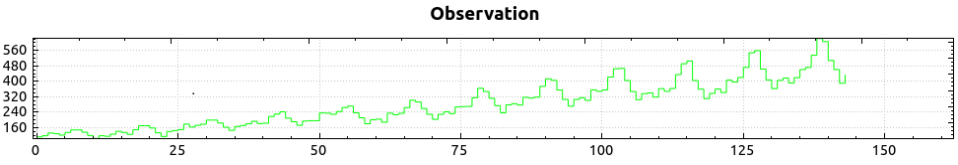

Figure 5.9: Non-stationary monthly passenger number data set.

The predictor is first applied to this unaltered series by observing the first 124 data points and then predicting the following 20 using both single-step and multi-step prediction. Then, the series is differenced using the method described in Section 2.2.4, resulting in a new series with the linear trend removed (Figure 5.10).

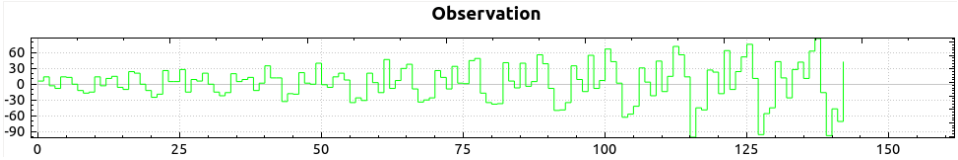

Figure 5.10: Differenced monthly passenger number data set.

The experiment is then repeated for the differenced series, using a signed predictor. This yields the results shown in Table 5.4. The predictor is clearly more capable of predicting the differenced series than the original series. This shows that for better results, the predictor requires a time series to be made stationary - which is not surprising given that the statistical techniques discussed in Section 2.2.1 have the same requirement.

	Single-Step		Multi-Step	
	RMSE	**NRMSE**	**RMSE**	**NRMSE**
Unaltered Series	90.0197	0.4422	101.3573	0.8760
Differenced Series	82.7726	0.2839	51.5650	0.1819

Table 5.4: The RMSE and NRMSE when predicting the unaltered and differenced airline passenger data set.

Using Weka, we find that with the SVM approach an excellent performance can be achieved on the non-stationary airline passenger data set. As shown in Figure 5.11, support vector regression is able to model this series very well and make accurate multi-step predictions. These results imply that data mining techniques are more effective at forecasting non-stationary time series.

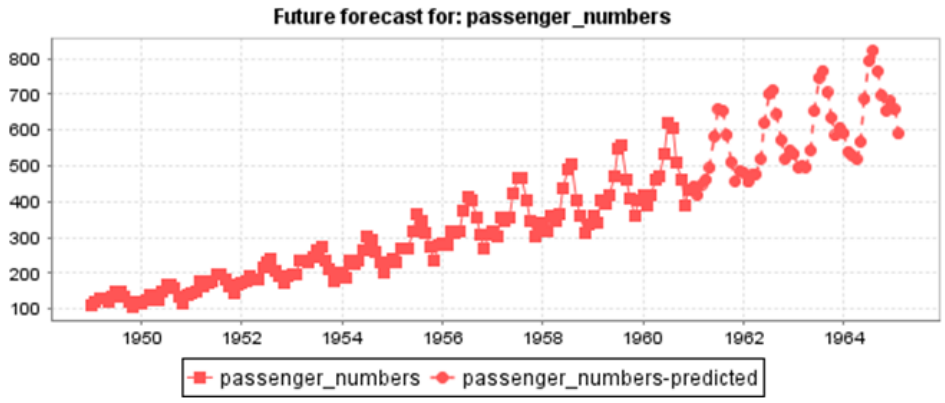

Figure 5.11: Using a data mining approach to forecast the non-stationary airline passenger number time series.

5.6 Comparison with Literature

A number of experiments have been carried out using synthetic and real-world chaotic time series with the aim of comparing the performance of ALPE with some of the recent chaotic prediction literature discussed in Section 2.5.

5.6.1 Synthetic Chaotic Time Series

Two versions of the Lorenz system are generated as described in Section 2.5.3. For this, a MATLAB implementation of the fourth order Runge-Kutta approximation technique is used. The generated data points are then scaled to work with a 16-bit predictor. To compare with the results found in [12] and [16], two experiments are carried out. For the first Lorenz system, 5500 data points are generated of which the first 3000 are discarded (to avoid a transient response), the next 1500 are passed to the predictor as observations and the last 1000 are used to test the predictor and measure our error. For the second Lorenz system, 5400 data points are generated of which the first 4800 are used in training and the following 600 are used in testing.

These experiments are then repeated on a noisy versions of the Lorenz system. For this, MATLAB is used to add Gaussian white noise to the series. For the first system, a zero mean Gaussian noise with a standard deviation of 5% of the standard deviation of the clean signal is added to

the clean signal. For the second system, white noise with a standard deviation of 0.1 is added to the clean signal. The results are reported in Table 5.5.

	ALPE		Literature	
	Noise-Free	**Noisy**	**Noise-Free**	**Noisy**
Lorenz System 1	0.1203	0.1300	3.13e-5	0.0256
Lorenz System 2	0.1633	0.2033	3.10e-4	0.0634

Table 5.5: NRMSE for single-step prediction of a noise-free and noisy Lorenz system.

We see that while ALPE is able to achieve reasonable predictive accuracy on both the noise-free and noisy Lorenz series, machine learning methods developed in the most recent research papers are able to achieve a much lower NRMSE when predicting the Lorenz system.

For further comparison, we perform a multi-step prediction on the first Lorenz system. This means the predictor attempts to predict all 1000 values just given those seen in training and without any correction to the actual values at each step. As reported in [12], we would expect the multi-step prediction to diverge rapidly due to the system's extreme sensitivity to initial conditions. However, interestingly we find that instead of diverging, the recursive prediction gets 'stuck' in a loop and makes the same set of predictions repetitively. This result is shown in Figure 5.12. This indicates that ALPE is not suitable for the multi-step prediction of time series which produce oscillating behaviour.

Next, we perform tests on the Mackey-Glass system in a similar fashion to the experiments carried out in [13]. The series is generated for $0 \leq t \leq 2000$ of which 1000 points between $t = 124$ to 1123 are kept for training and testing. Having scaled the data appropriately, a 20-bit predictor is trained on the first 500 observations and then the next 500 are used to evaluate the predictive accuracy. The predictions are then scaled back to their original form and the RMSE error metric is calculated. We found that ALPE achieved an RMSE of 0.069. When compared with the results of [13], we find that this is an improvement on the best auto-regressive model found for which the RMSE was 0.078. However, it was also shown that a fuzzy inference system could predict the same series with an RMSE of just

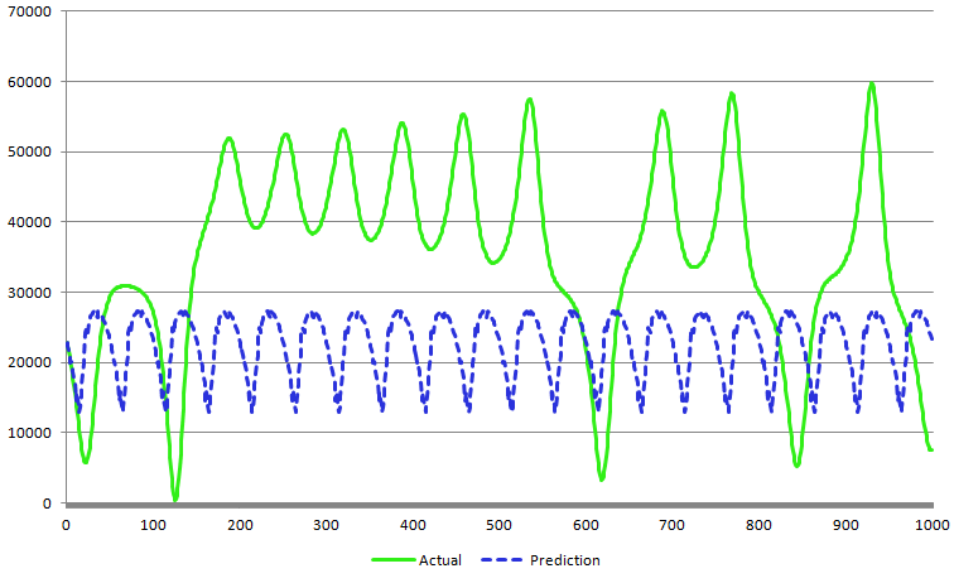

Figure 5.12: Prediction loop when making a recursive multi-step forecast for 1000 Lorenz series values.

0.0015. It should be noted that there is a significant disadvantage to the fuzzy inference system approach: their model took over 2 hours to build on an HP Apollo 700 Series workstation. This would perhaps be only a few minutes on today's computers, however our implementation of ALPE builds a model and makes its predictions in under ten seconds.

Finally, we experiment with the Henon map. The series is generated for $0 \leq t \leq 1000$ as described in Section 2.5.4. The first 900 observations are used in training, and error metric is computed over the remaining 100. The algorithm achieves an NRMSE of 0.3394. This shows that a reasonable model has been constructed for the Henon map, as we suspected in Section 5.2.3. When compared with the findings of [17], we see that this is much better than the exponential smoothing and ARIMA techniques which, when applied to the same data set, completely failed to model the chaotic behaviour - as they scored NRMSEs of 1.06 and 0.83 respectively. However, the evolutionary neural network approach is shown to be the most accurate when applied to the Henon map, with an NRMSE of 0.13.

5.6.2 Real-World Chaotic Time Series

The time series for the smoothed monthly sunspot number, discussed in Section 2.5.5, was downloaded from the SIDC (World Data Center for the Sunspot Index) [26].

A multi-step prediction of the 23rd solar cycle is then carried out, by passing all data up to January 1999 to the predictor and then predicting the next 26 months (corresponding to the 23rd solar cycle). Figure 5.13 shows that our predictions do not follow the actual series very closely, though the general trend is predicted correctly. When compared with Figure 2.7 we see that the neurofuzzy models from [12] make a better multi-step prediction. Their model has also estimated the solar cycle's peak (120.8) very accurately, whereas our predictions give an overestimation.

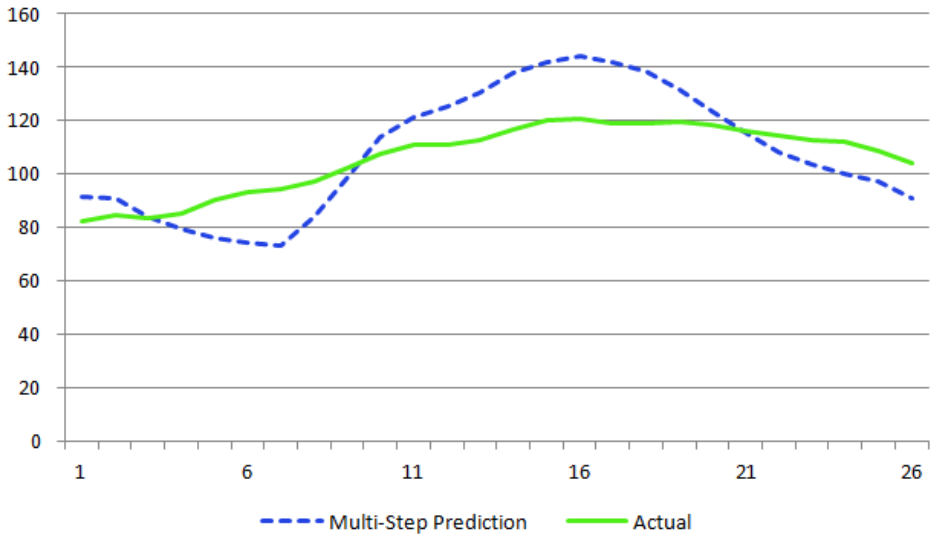

Figure 5.13: The predictor's recursive multi-step forecast for the 23rd solar cycle (January 1999 - 2001).

The predictor is now applied to problem of predicting daily river flow rates of the Mississippi river. Data for the daily river flow rates between the years of 1969-1987 are obtained from the US Geological Survey. Being real-world measured data, there were inevitably some missing data points. We were able to approximate these missing values using linear interpolation. The data was then scaled for compatibility with a 20-bit predictor, and the

first 15 years (5480 data points) were passed to the predictor for model construction. We then predicted the following two years (730 data points) using single-step prediction. As shown in Figure 5.14, the predictor's model proved to be very accurate for single-step prediction, with the occasional 'spike' where the prediction deviates from the actual value. This experiment was repeated using 3-step ahead prediction and 5-step ahead prediction and the NRMSE metric calculated for each.

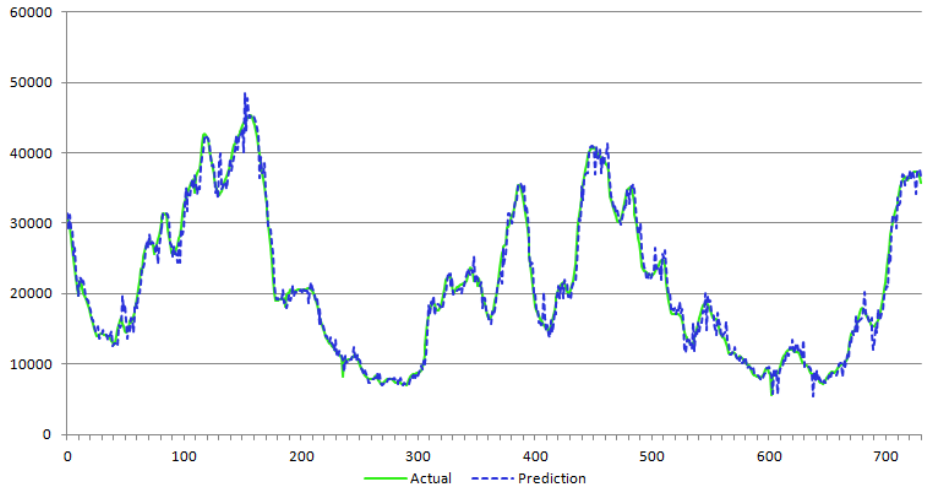

Figure 5.14: The predictor's single-step forecast for the daily flow rate of the Mississippi river over a two year period (1984-1986).

We compare our results with those of [16], where an artificial neural network is used to tackle the same prediction problem. The data from 1975-1993, as used in [16], was unfortunately unavailable and so while we use the same amount of training and test data in our experiments, it should be noted that the data itself is not identical. For all three experiments, our results and those from [16] are shown in Table 5.6.

	ALPE	**MLP**
1-step ahead	0.1197	0.0388
3-step ahead	0.2517	0.1330
5-step ahead	0.3635	0.2435

Table 5.6: Comparison of NRMSE of ALPE with the MLP approach when predicting Mississippi river daily flow rates.

Our results show that ALPE performs very well for the single-step prediction and makes reasonable predictions when predicting 3 and 5 steps ahead. However, the MLP appears to have achieved a lower NRMSE in all three experiments. Of course, this could be a result of differing data in the learning and test phases.

5.7 Application to Compression

Having observed the models produced by ALPE we decided to investigate whether the algorithm could be used in the domain of compression. When applied to time series, the algorithm builds a model to predict future values in the series. What if, instead of passing numeric observations to the learning algorithm, we pass ASCII characters or the bytes of a binary file? If the algorithm's model can represent the data accurately there is potential for compression, depending on the size of the model. Even if the model is not a perfect representation of the data, there is potential for lossy compression. To find out if the ALPE method can be applied in this way, we use the ASCII compression utility described in Section 4.4.6 to test the predictor's compression capability on some simple ASCII strings. The compression capability is evaluated by calculating the decompression accuracy (percentage of characters correctly predicted in the reconstructed string) and the size of the model. This analysis is performed twice: firstly using an 8-bit predictor where each ASCII character is treated as an observation and secondly using a single-bit predictor where the ASCII file is treated as a stream of bits. The strings used in testing are:

```
(1) abc (24 bits)
(2) aaabc (40 bits)
(3) agbyagagbyagbybyagbyagbybyagagby (256 bits)
(4) The quick brown fox jumps over the lazy dog (344 bits)
```

Table 5.7 shows the smallest model size (number of nodes) and minimum number of initial characters needed to decode the original string with *100% accuracy*, for each string.

We can see that the model size needed to represent a string with 100% accuracy is always less for the single-bit model than for the 8-bit model when using the same number of initial characters. Therefore the single-bit model

	8-Bit Model			1-Bit Model		
String	**Runs**	**Chars**	**Nodes**	**Runs**	**Chars**	**Nodes**
(1)	2	1	42	3	1	24
(2)	2	3	74	3	3	40
(3)	12	6	557	41	6	257
(4)	3	5	776	5	5	345

Table 5.7: Number of runs, minimum initial characters and minimum model size needed to decode each test string with 100% accuracy.

always gives a more optimal solution. However, the number of nodes required by the single-bit model is always approximately equal to the number of bits needed to store the original ASCII string - therefore no compression can be gained from using these models. During experimentation, it also becomes clear that a small decrease in the model size leads to a big decrease in accuracy. For example, if we decrease the number of runs used in (4) from 5 to 4, the accuracy drops from 100% to 44% and the number of nodes in the model drops from 345 to 335. In this case, a decrease in the model size of just 3% has resulted in a 56% drop in accuracy. This indicates that it is not feasible to apply the algorithm in its current form to compression problems.

6 Conclusions & Future Work

We have developed a fully functional and feature-rich C++ implementation of ALPE which has enabled us to conduct an in-depth investigation into the power, applicability and potential of this novel technique. By extending the original algorithm to handle observations and predictions of an arbitrary bit size, we were able to apply the algorithm to a diverse range of time series, as long as the series were first scaled appropriately. In this chapter, we present a summary of our results and discuss the conclusions we can draw from them.

Our initial experimentation showed that, statistically, the algorithm constructs a good model for many series, as no significant evidence for the autocorrelation of residuals was found. We saw that the algorithm demonstrates the potential to make single-step predictions for certain chaotic series with remarkable accuracy. This is especially true for the chaotic logistic map and the Henon map - two non-linear chaotic series which are known to be very difficult to predict. The algorithm's ability to learn from the very first observation and to discover complex patterns within a short space of time is particularly impressive. The best example of this is for the chaotic logistic map, where ALPE is able to construct an accurate phase space representation after observing under 200 data points.

When artificial white noise was added to the data we found that the algorithm was still able to construct a reasonable model for both simple and chaotic series - however, the noise had a larger effect than expected. We found that when noise is added to the periodic sine wave there is clear autocorrelation in the residuals - indicating that in this case the algorithm is not building the best possible model for the observed series.

Our experiments with the Weka machine learning framework demonstrate that the algorithm is able to outperform the basic MLP and SVM on certain series, such as the sine wave and chaotic logistic map. However, this is not always the case. In particular, for the Mackey-Glass series, the MLP and

SVM yield more accurate predictions. We also saw that ALPE performs best on stationary data - where any linear trend has been removed. When applied to a non-stationary series, Weka's SVM approach showed the ability to model the series with considerably higher accuracy. This was not a surprising result, given that statistical prediction techniques, such as ARIMA, share the same requirement of stationary data.

For the Lorenz system we observed similar results to those found with the Mackey-Glass series. Other techniques were able to give a better performance on both noise-free and noisy versions of the system. Interestingly, it seems that ALPE performs best on chaotic series derived from difference equations (e.g. the logistic map and Henon map) rather than those derived from differential equations, such as the Mackey-Glass series and Lorenz system.

When applied to real-world data series, we saw that ALPE was able to achieve impressive accuracy for single-step prediction. This was demonstrated by its accurate predictions of the Mississippi river daily flow rate. However, for multi-step prediction of the sunspot number series we found that techniques developed in recent literature give a significantly better performance.

From our own analysis, as well as our comparison with recent literature, the MLP and adaptations of it appear to present the strongest competition in the field of time series prediction at present. When choosing a prediction technique, there are many aspects to consider including preprocessing steps, configuration requirements, accuracy, model size and the speed and complexity of the learning procedure. Many of these key differences between ALPE and the typical MLP are summarised in Table 6.1.

Our work in developing a compression analysis utility and applying the algorithm to the problem of compression proved to be an interesting experiment. We saw that the models generated by the algorithm can be used to encode a bit stream and later decode the original data or a lossy approximation of it. However, the models were consistently too large for the algorithm to have any useful application in this domain.

In summary, we have shown that ALPE is a powerful, online, non-parametric learning technique which can be applied successfully to a diverse range of stationary time series. It's performance on chaotic time series is particularly impressive and it is able to outperform state-of-the-art statistical and

machine learning methods in certain cases. When compared with other techniques which require careful parameter configuration and consist of separate training and testing phases, ALPE's speed of learning and easy configuration stands out as being significantly advantageous.

Listed below are some ideas for future work and investigation on the algorithm and C++ implementation.

- Determine the reason for apparent anomalies when predicting the Mackey-Glass and Lorenz systems. These are commonly the cause of higher than expected prediction errors.
- Investigate the cause of cyclic behaviour in multi-step prediction. Adapt the algorithm to avoid prediction loops.
- Investigate the potential of producing more compact and efficient models. If successful, re-evaluate the algorithm's applicability to compression.
- Alter the C++ implementation to automatically detect the required bit size needed to represent a given series - to avoid the need to set this manually.
- Extend the C++ implementation to include functionality for additional preprocessing, such as differencing and automatic scaling. Again, this would reduce the manual steps which are currently necessary to analyse certain time series.

While the algorithm analysed and evaluated in this report is a very effective prediction technique, ongoing work to refine the algorithm and resolve issues including those mentioned above is being undertaken by Ben Rogers. The algorithm is currently protected under an NDA. For further information, Ben can be reached at ben.rogers1@virgin.net.

Characteristic	**ALPE**	**Typical MLP**
Data	Discrete observations and predictions.	Continuous observations and predictions.
Learning Style	Online, sequential learning - Every new observation contributes to the training of the model.	Offline learning - Separate training and testing phases, model is fixed after training phase.
Learning Speed	Fast and immediate, learning begins from the first observation; each observation is only observed once.	Slow training phase involving many iterations over the training data to make adjustments to neuron weights.
Parameters	Non-parametric - No model-specific parameters need to be set. Only general settings, such as predictor bit size and model size limit must be specified.	Parametric - Model-specific parameters including the number of hidden layers, size of hidden layers and bias parameters must be set.
Model Size	Continually growing (until limit) for noisy or real-world series. Fixed for series which can be perfectly modelled.	Fixed after training phase.
Stationarity	Works well for stationary time series - requires removal of linear trend.	Works well for stationary and non-stationary series.
Preprocessing	Data points must be scaled during preprocessing to integers of a target bit size. Additional preprocessing such as differencing may be required if the original series is non-stationary.	No scaling required but requires time dependency to be encoded as an additional input field.

Figure 6.1: List of key differences between the ALPE technique and a typical MLP.

Appendix

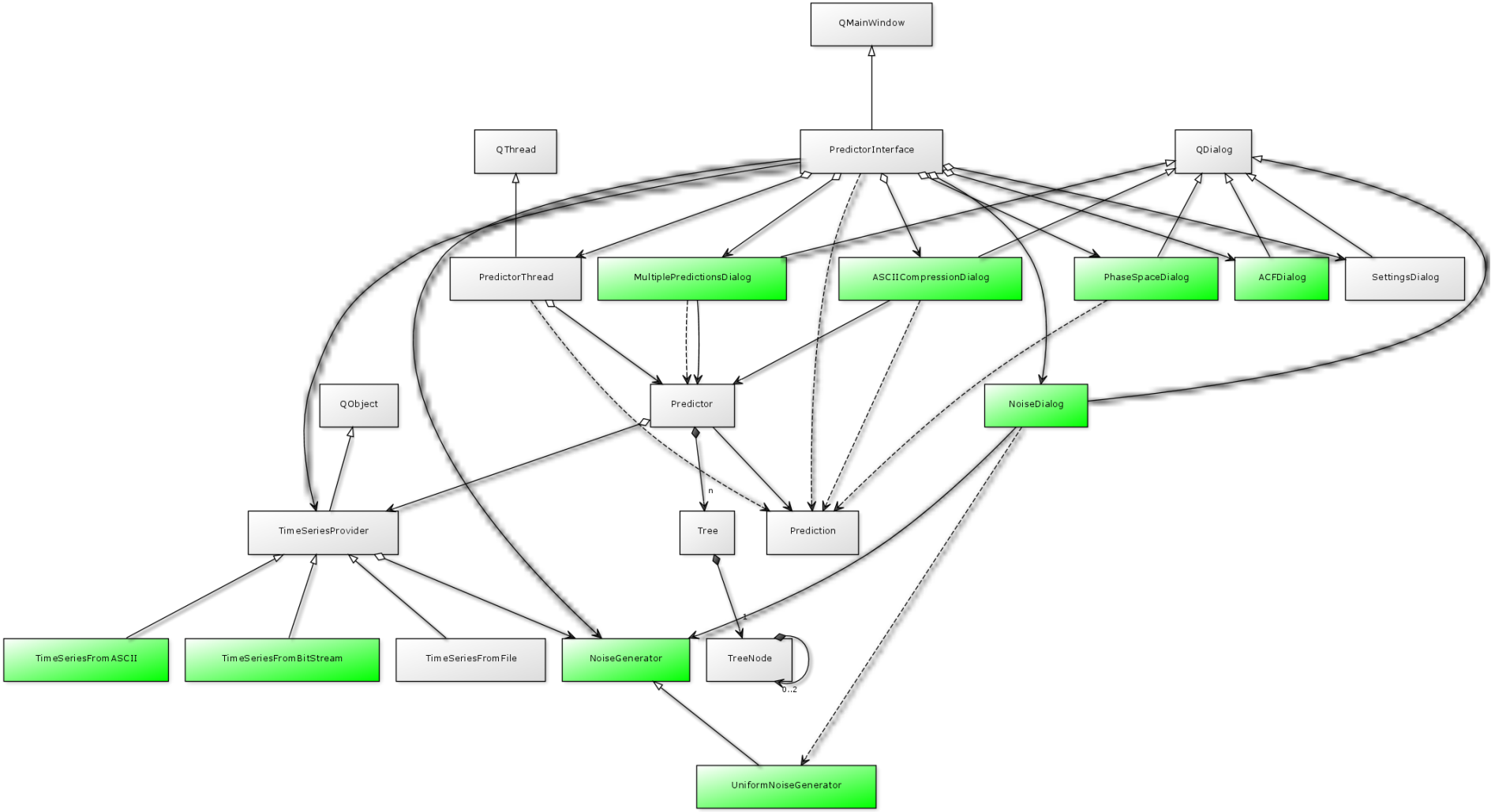

Figure A1: A complete UML diagram, showing all classes involved in the final implementation. Classes shaded in green are those which were added as part of the extensions discussed in Section 4.4.

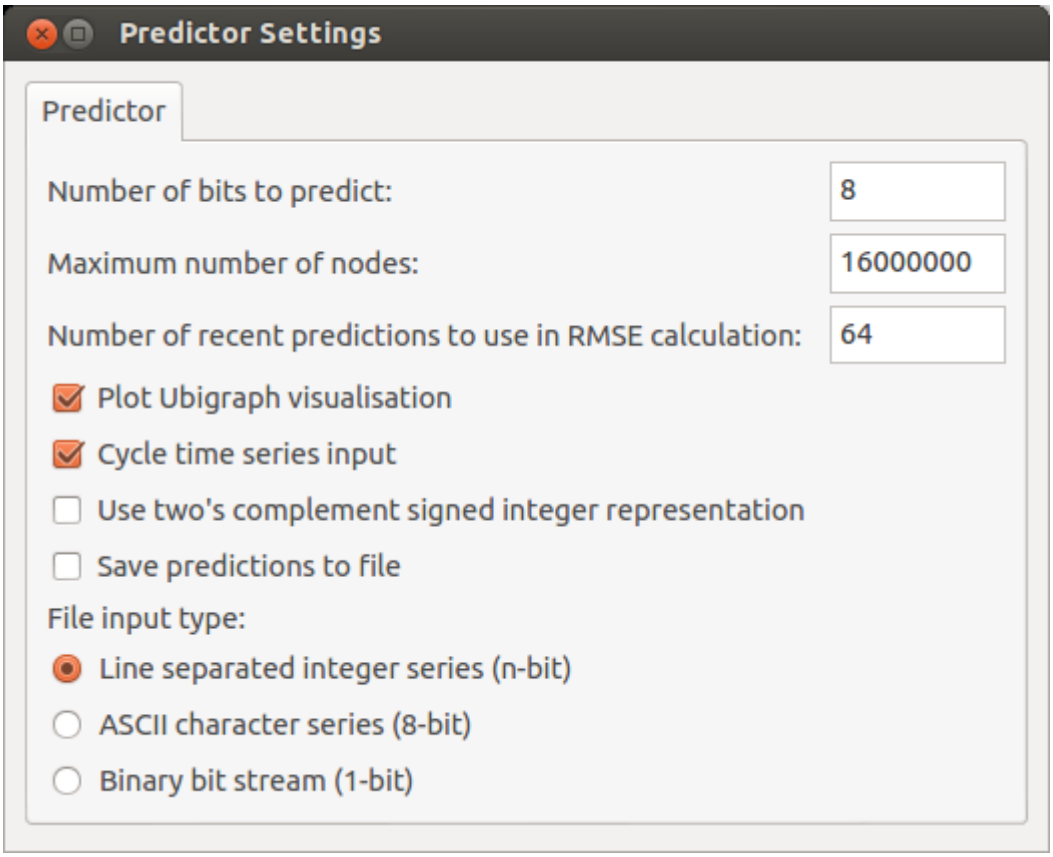

Figure A2: The settings dialog in the C++ implementation.

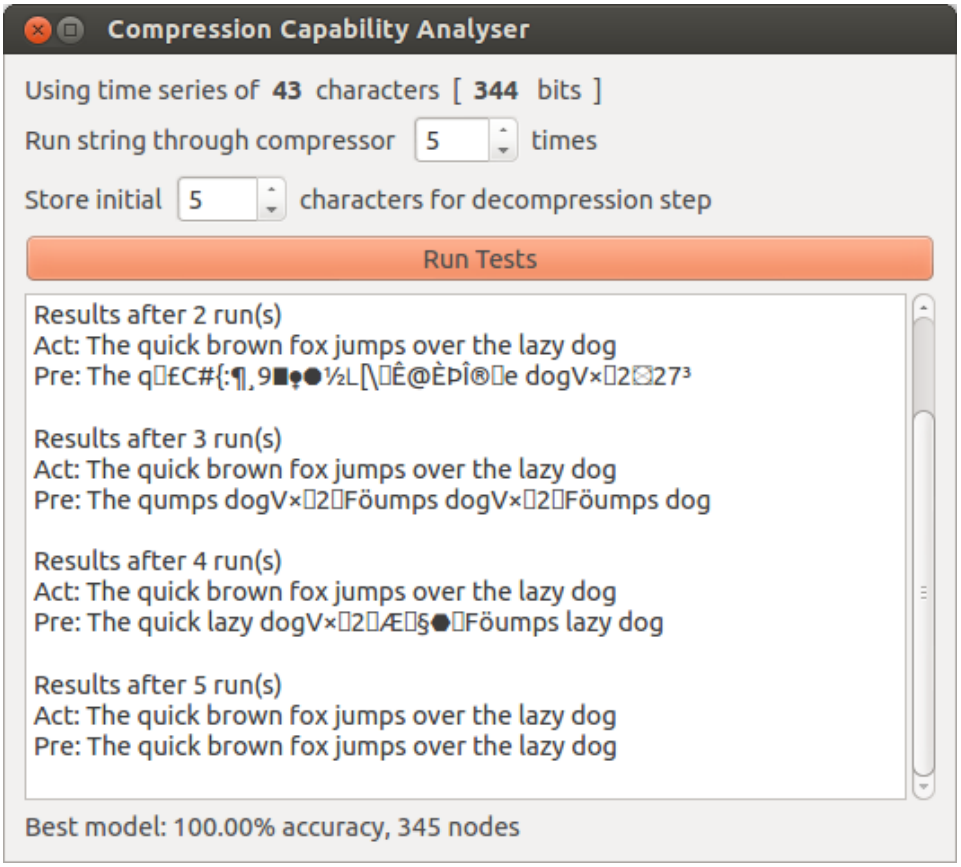

Figure A3: The compression analysis utility in action.

Bibliography

[1] P. Pai and C. Lin, "A hybrid arima and support vector machines model in stock price forecasting," *Omega*, vol. 33, no. 6, pp. 497–505, 2005.

[2] R. Lawrence, "Using neural networks to forecast stock market prices," *University of Manitoba*, 1997.

[3] B. Ernst, B. Oakleaf, M. Ahlstrom, M. Lange, C. Moehrlen, B. Lange, U. Focken, and K. Rohrig, "Predicting the wind," *Power and Energy Magazine, IEEE*, vol. 5, no. 6, pp. 78–89, 2007.

[4] A. Morales-Esteban, F. Martínez-Álvarez, A. Troncoso, J. Justo, and C. Rubio-Escudero, "Pattern recognition to forecast seismic time series," *Expert Systems with Applications*, vol. 37, no. 12, pp. 8333–8342, 2010.

[5] G. E. P. Box, G. M. Jenkins, and G. C. Reinsel, *Time Series Analysis, Forecasting and Control*. Englewood Clifs, NJ: Prentice Hall, 3rd ed., 1994.

[6] B.-S. Lin, D. MacKenzie, and T. Gulledge Jr, "Using ARIMA models to predict prison populations," *Journal of Quantitative Criminology*, vol. 3, pp. 251–264, 1986.

[7] N. Saeed, A. Saeed, M. Zakria, and T. M. Bajwa, "Forecasting of wheat production in pakistan using ARIMA models," *International Journal of Agriculture & Biology*, 2000.

[8] "Time Series Analysis and Forecasting with Weka." `http://wiki.pentaho.com/display/DATAMINING/Time+Series+Analysis+and+Forecasting+with+Weka`. Accessed: 15/01/2013.

[9] U. Thissen, R. Van Brakel, A. De Weijer, W. Melssen, and L. Buydens, "Using support vector machines for time series prediction," *Chemometrics and intelligent laboratory systems*, vol. 69, no. 1, pp. 35–49, 2003.

[10] D. M. Rice, "Predictability of outcomes: chaos theory and diabetes education," *The Diabetes Educator*, vol. 33, no. 1, pp. 31–32, 2007.

[11] H. Peitgen, H. Jürgens, and D. Saupe, *Chaos and fractals: new frontiers of science*. Springer, 2004.

[12] A. Gholipour, B. Araabi, and C. Lucas, "Predicting Chaotic Time Series Using Neural and Neurofuzzy Models: A Comparative Study," *Neural Processing Letters*, vol. 24, pp. 217–239, 2006.

[13] J. Jang and C. Sun, "Predicting Chaotic Time Series with Fuzzy If-Then Rules." University of California Berkeley, 1993.

[14] M. Mackey and L. Glass, "Oscillation and chaos in physiological control systems," *Science*, vol. 197, p. 287, 1977.

[15] H. Abarbanel, *Analysis of Observed Chaotic Data*. 1996.

[16] D. Karunasinghe and S. Liong, "Chaotic time series prediction with a global model: Artificial neural network," *Journal of Hydrology*, vol. 323, pp. 92–105, 2006.

[17] P. Cortez, M. Rocha, and J. Neves, "Evolving time series forecasting neural network models," 2001.

[18] Q. Zhang, "A nonlinear prediction of the smoothed monthly sunspot numbers," *Astronomy And Astrophysics*, vol. 310, pp. 646–650, 1996.

[19] S. Liong, K. Phoon, M. Pasha, and C. Doan, "Efficient implementation of inverse approach for forecasting hydrological time series using micro ga.," *Journal of Hydroinformatics*, vol. 7, pp. 151–163, 2005.

[20] X. Yu, S. Liong, and V. Babovic, "Ec-svm approach for real-time hydrologic forecasting.," *Journal of Hydroinformatics*, vol. 6, pp. 209–223, 2004.

[21] J. Farmer and J. Sidorowich, "Predicting Chaotic Time Series," *Physical Review Letters*, vol. 59, no. 8, 1987.

[22] S. Sello, "Solar cycle forecasting: a nonlinear dynamics approach," *Astronomy And Astrophysics*, vol. 377, pp. 312–320, 2001.

[23] M. Aly and H. Leung, "Chaotic Time Series Prediction Using Data Fusion." International Conference on Data Fusion, 2001.

[24] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten, "The weka data mining software: an update," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[25] "Henk Bruin's Research Interests." `http://http://personal.maths.surrey.ac.uk/st/H.Bruin/res.html`. Accessed: 08/06/2013.

[26] "World Data Center for the Sunspot Index." `http://sidc.oma.be/sunspot-data/`. Accessed: 31/05/2013.