INNOVATIONS IN LOCAL MODELING FOR TIME SERIES PREDICTION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

James McNames

May 1999

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Bernard Widrow
(Principal Adviser)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jerome H. Friedman
Department of Statistics

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jonathan P. How
Department of Aeronautics and Astronautics

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

Abstract

Previous studies have shown that local models are among the most accurate methods for predicting chaotic time series. This work discusses a number of improvements to local models that reduce computation, improve the model accuracy, or both.

Local models are often criticized because they require much more computation than most global models to calculate the model outputs. Usually, most of this time is taken to find the nearest neighbors in the data set. This work introduces two new nearest neighbor algorithms that drastically reduce this time and enable local models to be evaluated very quickly. The two new algorithms are compared with fifteen other algorithms on a variety of benchmark problems.

Local linear models are the most popular, and often the most accurate, type of local model. However, using an appropriate means of regularization to eliminate the effects of an ill-conditioned matrix inverse is crucial to producing accurate predictions. This work describes the two most popular types of regularization, ridge regression and principal components regression, and two new generalizations of each of these methods that enable more accurate models to be constructed.

The accuracy of local models is sensitive to the values chosen for the model parameters. Most researchers pick the values for these parameters based on their experience and intuition. In this work, new optimization algorithms are introduced that improve the model accuracy by adjusting the initial parameter values provided by the user. These iterative algorithms take advantage of local models' ability to efficiently calculate the leave-one-out cross-validation error, an excellent measure of model accuracy that does not cause overfitting.

When local models are used to predict chaotic time series, there are several improvements that can be made that use the properties of chaotic systems to generate more accurate

predictions. This work describes several new modifications of this type and demonstrates their effectiveness on a variety of chaotic time series.

Acknowledgements

I am grateful to all of the friends and acquaintances that have helped me along the lengthy journey through graduate school. I would especially like to thank the following people who have influenced my thinking and values along the way.

Mahmood Nahvi and Ali Shaban of California Polytechnic State University served as initial role models and inspired me to pursue graduate school and, eventually, teaching.

Gregory Plett provided invaluable assistance with LaTeX and the mechanics of reaching graduation. The reliable Dr. Mike spent many hours repairing my damage to Simoon and has never hesitated to provide precise and helpful answers to my many questions on UNIX and neural networks. I would also like to thank other former members of the Zoo including Eric Wan, Edward Plummer, Raymond Shen, Françoise Beaufays, and Jeff Wilkins and current member Max Kamenetsky.

Michael Godfrey spent many hours discussing my research and career goals. He has also helped to guide me along a more direct path to graduation. Jerry Friedman profoundly influenced my understanding and thought on the problems addressed in this work and continues to serve as a role model for my career. Jonathan How has helped shape my thinking on dynamic systems and has opened my eyes to the interesting problems of control.

Bernard Widrow has ingrained in me the importance of improving the world and helping people through my research and career; for this I am deeply indebted. I also wish to thank him for introducing me to the world of adaptive signal processing, for providing a strong emphasis on solving practical problems, and for his thorough and thoughtful review of this work.

I would not have made it to graduation without the support and assistance of Joice DeBolt. I am especially grateful for the countless times she assisted me in times of crisis.

Marianne Marx enabled me to support myself as a teaching assistant during my time at Stanford. I am grateful for the many opportunities to instruct students and all that I have learned from them.

Although I have never met Tim Sauer, his thoughtful and successful work with local models for time series prediction provided the initial inspiration for this work. Other researchers who significantly influenced my thought during this work include Edward Lorenz, Martin Casdagli, J. Doyne Farmer, John Sidorowich, and Dimitris Kugiumtzis.

Although the internet has made it easy for researchers to publicly distribute data, most researchers in chaotic systems have chosen not to. I would like to thank those who have, including Andreas Weigend, Nick Tufillaro, Eric Weeks, Scott D. Bartholoma, and Rob Hyndman. I would especially like to thank Tom Carroll for personally giving me some of his chaotic circuit data.

Johan Suykens hosted the Lueven time series prediction competition, which proved to be a useful forum for comparing my ideas with other researchers. I benefited greatly from discussions with him and the participants at the workshop, most notably Patrick McSharry and Gianluca Bontempi.

Bill Staib, a friend and former employer, provided the opportunity to implement many statistical models that gave me a solid understanding of nonlinear modeling methods and their shortcomings. Mark Dzwonczyk, my current employer and friend, gave me the means to support my family and the flexibility to complete this work.

I am grateful to Tad Shannon, Donna McNames, and Hope Del Carlo for reviewing Chapter 3.

My parents, Donna and Robert McNames, provided constant support and encouragement throughout this work. Finally, I wish to thank my beloved friend and wife, Hope Del Carlo, for her persistent support, warm meals, and enduring love through many rainy nights in front of the computer.

This dissertation is dedicated to the memory of Etta Duran, Jess McNames, Jerry Morrison, and those devoted to the pursuit of peace on Earth.

Contents

List of Tables

List of Figures

List of Algorithms

Notational Conventions

In this work, the size of the vectors and matrices is denoted by the superscripts of $\mathbb{R}$. For example, if a matrix A contained i rows and j columns, this would be denoted as $A \in \mathbb{R}^{i \times j}$. Similarly, if a vector q contained i elements, this would be denoted as $q \in \mathbb{R}^{i}$. The transpose of vectors and matrices is denoted as q^{T}. The product of a symmetric matrix with itself is denoted as $W^2 = W^{\mathrm{T}}W$.

Vectors are indexed by subscripts; for example, q_i represents the ith element of the vector q. Occasionally, a vector will already have a subscript and a second subscript is used to represent the elements of the vector. For example, $x_{i,j}$ represents the jth element of the vector x_i.

An $i \times j$ matrix containing ones is denoted as $\mathbf{1}_{i,j}$. If the second subscript is omitted, it is assumed to be one so that $\mathbf{1}_k$ is a column vector with k elements.

The notation $O(n)$, pronounced "order N", describes the relative order of computation required to perform an operatation. More formally, if $\mathrm{C}(n)$ is a measure of the amount of computation required to perform an operation where n is a parameter of the operation, the computation is said to scale as $O(n)$ if there are positive constants c and n_0 such that $\mathrm{C}(n) \leq cn$ when $n \geq n_0$. A more detailed discussion of this notation is given in Weiss [1].

Variables

δ	Embedding delay.
γ	A vector that includes all of the model parameters.
$\phi_{\max}$	The maximum value of the amplification factors in a semi-global line search.
ω	Time series window length.
ν	Vector of the linear model coefficients.
A	The data matrix that contains the k neighboring points and a column of ones. $A \in \mathbb{R}^{k \times n_d+1}$.
b	Vector of the target values for each of the k neighboring points.

d	Vector that contains the sorted distances to the nearest neighbors such that d_i is the distance to the ith nearest neighbor.
I	The identity matrix.
k	The number of neighbors.
n_a	Number of steps ahead (for time series prediction).
n_c	Number of points used to calculate the cross-validation error.
n_d	Dimension of the observed vectors, $x, q \in \mathbb{R}^{n_d}$.
$n_{i,\max}$	The maximum number of iterations allowed for the generalized cyclic coordinate method to converge.
n_ℓ	Number of local model inputs.
n_p	Number of points in the data set.
n_ϕ	Number of amplification factors used in a semi-global line search.
$n_{r,\max}$	The maximum number of times the range of the semi-global line search may be reduced in the generalized cyclic coordinate method.
n_σ	The number of principal directions. Used with principal components regression.
n_u	The number of gradient-based parameter updates per a cycle of the generalized cyclic coordinate method.
n_y	Number of points in a time series, $\{y_1, \ldots, y_{n_y}\}$.
q	Query vector of observed variables. This vector is not included in the data set, $q \neq x_i \, \forall i$.
s_c	The center of the threshold transition region for principal components regression with soft thresholds.
s_w	The relative width of the threshold transition region for principal components regression with soft thresholds.
x_i	A vector of observed variables that is part of the data set.
y	The process output. Sometimes this is written as a function, $y_i = y(x_i)$, to show explicit dependence on the process observed variables.

Functions

$\mathrm{BW}(d_i, d_{k+1})$	The biweight function.
$\mathrm{CVE}(\gamma)$	The cross-validation error as a function of the model parameters.
$\mathrm{D}(q, x_i)$	The distance between the vectors q and x_i. Sometimes a subscript will be used to distinguish different measures of distance.
$g(x_t)$	Takens' equivalent function.
$\hat{g}_\gamma(x_t)$	Approximation of Takens' function. The model parameters are stored in the vector γ.
$\mathrm{P}(\gamma)$	The total penalty as a function of the model parameters.

Operators

$\underset{\alpha}{\operatorname{argmax}} \, g(\alpha)$	Returns the argument that maximizes the function $g(\alpha)$.

$\underset{\alpha}{\text{argmin}}\, g(\alpha)$	Returns the argument that minimizes the function $g(\alpha)$.
$\text{diag}(\gamma)$	Converts the vector γ into a diagonal matrix where the diagonal is equal to the argument vector.
$\text{E}[\cdot]$	Expected value.
$\nabla_\gamma f(\gamma)$	The gradient of the scalar function $f(\gamma)$ with respect to the parameters in the vector γ.
$\nabla^2_\gamma f(\gamma)$	The hessian of the scalar function $f(\gamma)$ with respect to the parameters in the vector γ.
$\triangle^{-1}$	A unit delay. For example, $\triangle^{-1} y_t = y_{t-1}$.

Abbreviations

CVE	Cross-validation error.
DOPAT	Depth-only principal axis tree, a nearest neighbor algorithm.
MSE	Mean squared error.
NMSE	Normalized mean squared error.
OLS	Ordinary least squares, an equation for the linear model coefficients.
OSCVE	One-step cross-validation error.
MSCVE	Multi-step cross-validation error.
PARTAN	Parallel tangents, a conjugate gradient optimization algorithm.
PAT	Principal axis tree, a nearest neighbor algorithm.
PCA	Principal components analysis.
PCR	Principal components regression.
PCTR	Principal components threshold regression.
PDS	Partial distortion search, a nearest neighbor algorithm.
RNMSE	The square root of the normalized mean squared error.
SVD	Singular value decomposition.
TPCR	Truncated principal components regression.
. VRR	Vectored ridge regression.
WVRR	Weighted vectored ridge regression.

Chapter 1

Introduction

But what ... is it good for?
—Engineer at the Advanced
Computing Systems Division of IBM,
1968, commenting on the microchip.

This chapter introduces the problems addressed in this dissertation and gives an overview of subsequent chapters. Section 1.1 defines the three problems: the nonlinear modeling problem, the nearest neighbors problem, and the time series prediction problem. Each of these problems occurs in a variety of applications, some of which are described in Section 1.2.

This work focuses on local modeling as a solution to the nonlinear modeling problem and the time series prediction problem. Section 1.3 describes the motivation for choosing local models over other methods and the practical difficulty of attempting a comparative study.

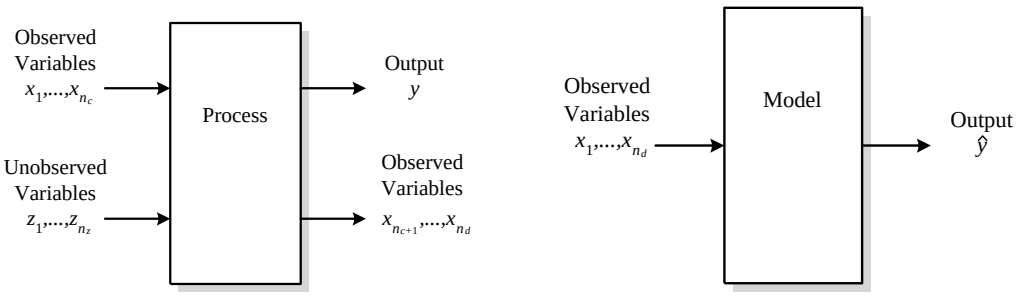
Section 1.4 summarizes a selection of the significant contributions made by the author. Section 1.5 concludes this chapter with an overview of the subjects addressed in each of the subsequent chapters.

1.1 Problem Definitions

This section defines the three problems addressed in this work. Each of the problems is discussed in more detail in subsequent chapters.

1.1.1 Nonlinear Modeling

The nonlinear modeling problem is defined as follows:



(a) Process (b) Model

Figure 1.1: (a) Illustration of the process from which the data set is collected. (b) The structure of the process model.

Definition 1.1

Consider a data set of n_p pairs of points, $\Omega = \{(x_1, y_1), (x_2, y_2), \cdots, (x_{n_p}, y_{n_p})\}$, where $x_i \in \mathbb{R}^{n_d}$ is a vector of observed variables and y_i is a scalar target variable. The nonlinear modeling problem is to estimate the corresponding target variable given an input vector, or query[1] *, of observed variables that is not part of the data set, $q \neq x_i \, \forall i$.*

This problem is sometimes called the nonparametric regression problem because the goal is to find the regression function of y on x. The term *nonparametric* means there is no known mathematical model that describes the relationship of y to x. This problem is also called the function approximation problem because the goal is to create a function that best describes the relationship of the variable y to the observed variables contained in the vector x, that is $\hat{y} = f(x)$.

The variable being estimated, y, is called the process output, the dependent variable, or the target variable. The vector of other variables, x, are called the observed variables, the independent variables, or the (model) input variables. A functional diagram of the source of the data set is shown in Figure 1.1*a*. The observed variables were split into two sets, one set at the input-end and the other at the output-end, to illustrate that the observed variables may not have a causal relationship with the process output. For example, if a model was to be constructed that estimated the concentration of nitric oxide in engine exhaust, the set of observed variables may be causal, such as fuel-air ratio, or non-causal, such as the

[1] *The term* query *is used throughout this work to mean an input vector that is not part of the data set. Sometimes the phrase "the model is queried" is used to mean an input vector is given to the model so that the model output can be calculated.*

exhaust manifold temperature. The model of the process, shown in Figure 1.1*b* uses both types of variables to estimate of the process output.

Here x_i is used to refer to input vectors in the data set. After the model is constructed, the target variable can be estimated by evaluating the model for input vectors that are not part of the data set. This is called querying the model and the vector q, a mnemonic for *query*, is used to represent the input vectors in this case[2].

The scalar y is used to refer to the process output for both points in the data set and points not in the data set. Occasionally, the model output is written in functional form, $\hat{y}(q)$, to explicitly show the dependence of the process output on the observed variables in contained in the vector q.

1.1.2 Nearest Neighbor Algorithms

The nearest neighbors problem is defined as follows:

Definition 1.2

Given a data set of n_p points, $\{x_1, x_2, \cdots, x_{n_p}\}$, find the k points that are closest to a point, q, where $q, x_i \in \mathbb{R}^{n_d}$ for all i. Here k is a user-specified parameter.

The obvious approach to solve this problem is to calculate the distance from the query point to each point in the data set. This approach, called the brute force method, may be acceptable for small data sets, but for large data sets the required computation can be excessive. Chapter 3 describes a number of fast nearest neighbor algorithms that can solve the problem much more quickly and two new algorithms are compared with leading algorithms on a variety of benchmark problems.

The nearest neighbor problem requires that a measure of closeness, or distance, be defined. Many distance measures have been used for local modeling and the nearest neighbor problem. This work focuses on the diagonally weighted Euclidean distance,

$$\begin{aligned} \mathrm{D}_{\mathrm{WE}}(q, x_i)^2 &= \sum_{j=1}^{n_d} \lambda_j^2 (q_j - x_{i,j})^2, \\ &= (q - x_i)^{\mathrm{T}} \Lambda^2 (q - x_i), \end{aligned}$$

[2]In machine learning the data set is called the training data and a collection of query vectors is called the test data.

where $\lambda \in \mathbb{R}^{n_d}$ is a vector of the metric weights and Λ is a diagonal matrix,

$$\begin{aligned}\Lambda &\triangleq \operatorname{diag}(\lambda), \\ &= \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_{n_d} \end{bmatrix}.\end{aligned}$$

By a transformation of variables, $\acute{q} \triangleq \Lambda q$ and $\acute{x_i} \triangleq \Lambda x_i$, the weighted Euclidean distance can be reduced to the unweighted Euclidean distance,

$$\begin{aligned}\mathrm{D_{WE}}(q, x_i)^2 &= (q - x_i)^{\mathrm{T}} \Lambda^2 (q - x_i), \\ &= (\acute{q} - \acute{x_i})^{\mathrm{T}} (\acute{q} - \acute{x_i}), \\ &= \mathrm{D_E}\,(\acute{q}, \acute{x_i})^2.\end{aligned}$$

This is useful because most nearest neighbor algorithms only work with the Euclidean distance (see Chapter 3); this technique enables these algorithms to work with the more general weighted distance. The unweighted Euclidean metric is also known as the L_2 norm and as the Minkowski metric.

1.1.3 Chaotic Time Series Prediction

The time series prediction problem is defined as follows:

Definition 1.3

Given a univariate time series of n_p points, $\{y_1, y_2, \cdots, y_{n_y}\}$, where y_i is a scalar, predict the next n_a points in the time series, $\{y_{n_p+1}, y_{n_p+2}, \cdots, y_{n_p+n_a}\}$.

The scope of this work is limited to time series prediction of deterministic nonlinear dynamic systems. If a few general assumptions are made about the system, the time series is chaotic and has several special properties. One of the most important of these properties is that a nonlinear autoregressive function exists that can predict the time series perfectly. The theorem that proves this, Taken's theorem, provides a theoretical foundation for framing the time series prediction problem as a nonlinear modeling problem.

The innovations in local modeling that are described in this work can be directly applied to the time series prediction problem. Further improvements are also described

that exploit the properties of chaotic time series to reduce computation and increase the accuracy of the predictions; these improvements are discussed in Chapter 6.

1.2 Applications

Each of the three problems addressed in this work occur in many different disciplines and there are many applications for the algorithms and techniques described in this work. This section describes a selection of these applications.

1.2.1 Nonlinear Modeling

Many different types of problems can be reduced to the nonlinear modeling problem including the time series prediction problem, pattern recognition, density estimation, and system identification.

There are many industrial, research, and commercial applications that use nonlinear models. These applications include image processing, speech recognition, time-series analysis, adaptive equalization, medical diagnosis, economic forecasting, and steelmaking [2–4].

1.2.2 Fast Nearest Neighbor Algorithms

The nearest neighbors problem is encountered in a wide range of applications including density estimation, pattern recognition [5], clustering [6], function approximation [7], time series prediction [8, 9], document retrieval [10, 11], optical character recognition [12], and vector quantization [6, 13]. In most of these applications the large computational cost of finding the nearest neighbors imposes practical limits on the data set size and the rate at which the application can operate. This has motivated the development of many fast nearest neighbor algorithms.

1.2.3 Chaotic Time Series Prediction

Applications that use prediction methods developed for chaotic time series include weather prediction [14–18], epidemiological forecasts [19, 20], population biology [19], astronomical analysis [18, 21–25], electrical circuit analysis [18, 26], fluid turbulence [18, 22, 27], flame dynamics [18], laser intensity prediction [28, 29], speech prediction and compression [18, 30], electroencephalogram (EEG) prediction [18, 27], heart beat (EKG) monitoring [20, 26], ocean wave forecasting [18], noise cancellation [30], mechanical vibration analysis [18, 31], climate prediction [18], and analysis of chemical reactions [32].

1.3 Motivation for Local Modeling

Local models postpone the computation required to construct the model until the model is queried. Once the input vector is available, the nearest neighbors in the data set are located and a simple model is constructed using only the neighboring points. After the model is constructed, it is evaluated using the input vector to produce the local model output.

These models are called *local* because the model is constructed using only a small neighborhood of points. In contrast, the construction of global models requires, or utilizes, the entire data set.

This section describes the problem of expert bias and discusses previous comparative studies between local models and global models. The strong performance of local models in these studies motivated this work.

1.3.1 Expert Bias

All of the leading nonlinear modeling methods require the user to make some decisions about the model architecture and to assign values to one or more parameters. Local models require the user to choose the size of the neighborhood, the type of distance metric, the weighting function, and the local model type[3]. Global models also require the user to make many decisions. For example, perceptron neural networks require the user to pick the network architecture, the method of weight initialization, the measure of model performance, and the optimization, or training, algorithm.

The accuracy of nonlinear models is usually sensitive to these user-specified parameter values and, therefore, the skill and knowledge of the user[4]. This is sometimes referred to as *expert bias*, though it could be just as aptly called *novice bias*.

The dependence of model accuracy on the skill of the user makes it difficult to conduct a convincing comparative study. This problem is exacerbated by the numerous variations on each type of model that have been described in the literature; all of which reportedly improve the model performance. The credibility of comparative studies is also called into question when the author favors a particular method.

For these reasons, this work does not include a comparative empirical study that compares the performance of the new methods to other types of nonlinear models.

[3]These concepts are introduced in Chapter 2.

[4]See Friedman [33] for a thorough discussion of this issue.

1.3.2 Local versus Global Modeling

Local models have performed very well in comparative studies on time series prediction problems and in most cases have generated more accurate predictions than global methods [18, 34–37]. However, since each of these studies was performed by no more than a few researchers, these results are subject to the problem expert bias. This problem is largely circumvented by competitions that bring together a large group of researchers to compare their expertise and best modeling methods on a common set of problems.

Two of these competitions are of special significance to this work. A time series prediction competition was held by the Santa Fe institute in 1991 [28]. Although several types of time series analysis were included in the competition, the prediction of a chaotic time series produced by a laser received the most attention and entries. The winner of the time series used a novel neural network architecture [38]. The second place entry, generated by a local linear model, was nearly as good [37]. A further comparison of these two methods was performed after the competition. On other segments of the time series the local model performed better than the neural network in three out of four trials[5]. Although the scope of the competition was too narrow to conclusively determine that any specific type of nonlinear modeling is best, the results suggest that local models are capable of performing at least as well as other leading nonlinear models.

A second competition was held in Leuven, Belgium in 1998 to assess the changes and improvements that had occurred in the field of time series prediction since the Santa Fe competition. Entrants were given a time series consisting of 2,000 points and were asked to predict the following 200 points. The competition was won by the author using a local model. The second place entry was also generated by a local model. In fact, only local models were able to forecast the first 80 steps accurately. Although the scope of this competition was also narrow in that only a single time series was used, the strong showing of local models strongly supports the claim that local models are among the best techniques for time series prediction [39, 40].

There are several distinct advantages that local models have over global models. These are discussed in Chapter 2 after a more thorough description of local models.

[5] This competition is discussed further in Chapter 7.

1.4 Contributions

The significant contributions of the author include the following.

1. **Two Fast Nearest Neighbor Algorithms:** This work describes two new nearest neighbor algorithms that incorporate a novel search tree based on principal components analysis. These algorithms use new bounding criteria to eliminate points during the search and reduce computation.

2. **Comparative Study of Fast Nearest Neighbor Algorithms:** A thorough comparative study of leading nearest neighbor algorithms is given in Chapter 3. The two new algorithms are compared to fifteen other nearest neighbor algorithms on common benchmark problems including synthetic data, image vector quantization data, speech data, and chaotic time series. This is the most thorough comparative study to date.

3. **New Method of Principal Components Regression:** Principal components regression (PCR) is a popular method of regularization for local linear models. This work describes several important generalizations of this method. First, the method of centering is generalized for use with weighted least squares. Second, a new method of using soft thresholds to decrease the influence of small singular values is described that ensures the model output remains smooth. Third, the method of using a truncated summation of singular values is combined with the soft threshold technique to form a new composite form of PCR.

4. **Cyclic Coordinate Optimization for Local Modeling:** Most researchers consider critical local model parameters, such as the neighborhood size (k), to be user-specified variables. This makes the accuracy of the model sensitive to the expertise of the user. In this work an attractive alternative is described that iteratively optimizes the critical model parameters so as to minimize the cross-validation error.

   Most iterative optimization algorithms require the gradient of the error function with respect to the model parameters. However, the gradient with respect to some of the local model parameters can not be calculated. As an alternative, the cyclic coordinate method is generalized so that some parameters can be optimized without the gradient and the others can be efficiently optimized using a partial gradient. Since the cross-validation error is used to measure the model accuracy, the problem of overfitting is avoided.

5. **Gradient-Based Optimization for Vectored Ridge Regression:** This work describes two generalizations of ridge regression. First, ridge regression is generalized for use with weighted least squares. Second, the ridge coefficient is replaced with a vector that applies a different penalty to each model input.

   For efficient parameter optimization, the gradient of the cross-validation error with respect to the metric parameters, model inputs, and ridge coefficients is derived. This enables many of the model parameters to be optimized by efficient gradient-based optimization algorithms.

6. **Multi-Step Cross-Validation Optimization for Iterative Models:** It is generally accepted that iterative models produce the most accurate multi-step predictions. However, when models are optimized to predict one-step ahead they often produce poor predictions because the one-step predictions do not take into account the errors in the input variables that occur with iteration. Two new algorithms are described that optimize the multi-step cross-validation error to account for this effect. The improved performance is demonstrated on chaotic time series from a wide variety of sources.

1.5 Thesis Overview

This chapter defined the three problems addressed in subsequent chapters: the nonlinear modeling problem, the nearest neighbors problem, and the time series prediction problem. This chapter also described some applications that involve each of these problems and provide the motivation for investigating local modeling.

Chapter 2 introduces local models and gives some examples to illustrate the consequences of choosing different local model types, distance metrics, and weighting. After the reader has gained a mathematical and intuitive understanding of local modeling, some of the qualitative tradeoffs between local modeling and global modeling are discussed.

Chapter 3 reviews some of the common techniques used in fast nearest neighbor algorithms to reduce computation. Two new algorithms are described and compared to a large selection of other leading methods on several types of benchmark problems. The use of fast nearest neighbor algorithms greatly reduces the computation necessary to use local models on large data sets.

Local linear models require the calculation of a matrix inverse during model construction. Because the neighborhood of points is often small, the matrix is often nearly singular

and the model output amplitude can be unreasonably large. This problem can be alleviated by using an appropriate method of regularization. Chapter 4 describes the two most popular methods of regularization and generalizes each of them in several important ways. The two new methods of regularization allow local linear models to be used with local weighting and provide the user with more control than previous methods.

Chapter 5 describes a new method of choosing values for critical model parameters based on an iterative minimization of the cross-validation error. Complete optimizaiton algorithms are given for the two new methods of regularization.

Chapter 6 describes several modifications to the methods of local modeling described in previous chapters. These modifications are specifically designed for the time series prediction problem and increase the model accuracy by exploiting the properties of chaotic time series. Several case studies of applying the new method of local modeling to chaotic time series is given in Chapter 7.

Chapter 8 summarizes this work and describes several ideas for further research.

Chapter 2

Introduction to Local Modeling

Computers in the future may weigh
no more than 1.5 tons.
—Popular Mechanics, forecasting the
relentless march of science, 1949

This chapter introduces local modeling and provides the mathematical foundation and motivation for the innovations described in the subsequent chapters.

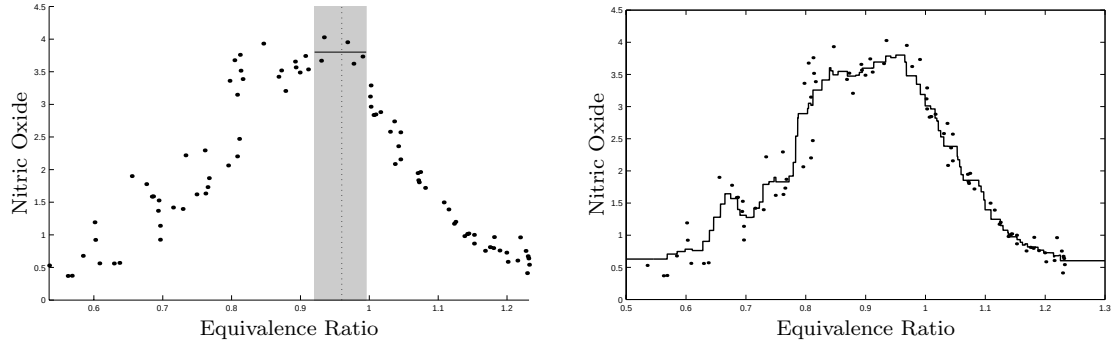
This chapter begins with a simple example that describes the process of building a local model. This is followed by a discussion of various measures of distance in Section 2.2 and of the two most popular types of local models in Section 2.3. This is followed by a description of local weighting in Section 2.4, a method for smoothing the outputs of local models. Section 2.5 illustrates some of the properties of local models using a data set with two input variables. Section 2.6 describes some of the problems that occur when local models are applied to problems that have more than two input variables and Section 2.7 lists some of the tradeoffs between local modeling and global modeling.

2.1 An Example of Local Modeling

Local models require a data set with many examples of both the observed variables and the process output. The model can then be constructed and used to estimate the process output for a new model input vector q, called a query, that is not part of the data set.

Local models usually defer most of the processing until the model is queried[1]. At that time the local model output is calculated using three consecutive stages. The first stage

[1]Because the model construction is delayed until the query vector is made available, local modeling is also referred to as *lazy learning* in the literature.



(*a*) (*b*)

Figure 2.1: (*a*) Illustration of how the local average is calculated. The query at 0.96 is shown by the dashed vertical line and the five nearest neighbors of the input are shown by the gray region. The horizontal line in the gray region shows the average of the five nearest neighbors. (*b*) Illustration of local averaging applied over the entire input domain.

finds the k closest vectors, or nearest neighbors, of observed variables in the data set. The second stage constructs a simple model using only the k nearest neighbors. The third stage evaluates the model using the query vector as the input to estimate the process output, $\hat{y}$.

For the sake of simplicity, two nonlinear modeling problems are used to introduce local modeling in this chapter. The first problem is to build a model that estimates the concentration of nitric oxides in exhaust from an automobile engine burning ethanol. The only observed variable, or model input, is the equivalence ratio. The corresponding data set, henceforth referred to as the Ethanol data set[2], consists of 88 points and is shown in Figure 2.1*a*. The second nonlinear modeling problem is described in Section 2.5.

To introduce local modeling, a simple example using the Ethanol data set is given. Suppose we wish to use a local model to estimate the output of the process for an input value of 0.96. If we choose to use five neighbors to build our local model, then the local model includes the points shown in the shaded region of Figure 2.1*a*. The location of the input is shown by the dashed vertical line. If we calculate the average of the five nearest neighbors and use this as our estimate, the model output is 3.80, as shown by the intersection of the horizontal and vertical lines in the shaded region. Note that none of the data set points outside of the shaded region affect the model output.

[2]This data set is briefly described in [41] and is available at *http://www.stern.nyu.edu/∼jsimonof/-SmoothMeth*.

Figure 2.1*b* shows the result of repeating this process for inputs taken over the entire domain of the data set. This shows what the function would look like if the inputs were continuous. Note that the local model output is not a continuous function of the model input; rather, it is only piece-wise constant. This is a common criticism of local modeling and will be addressed in Section 2.4.

2.2 Selection of a Distance Metric

Most researches choose the Euclidean metric, introduced in Chapter 1 as the measure of distance between two points,

$$\mathrm{D}_{\mathrm{E}}\left(q, x_i\right) = \left(\sum_{j=1}^{n_d} (q_j - x_{i,j})^2 \right)^{1/2}, \tag{2.1}$$

where q is the query vector and x_i is a point in the data set.

Many other measures could be used. For example, any p-norm could have been used, as defined by

$$\mathrm{L}_p(q, x_i) \triangleq \left(\sum_{j=1}^{n_d} |q_j - x_{i,j}|^p \right)^{1/p},$$

where the norm parameter p is chosen by the user. Most researchers prefer the Euclidean distance, though $\mathrm{L}_\infty(\cdot,\cdot)$ and $\mathrm{L}_1(\cdot,\cdot)$ are also common choices.

Very little research has been done to determine which measures of distance make local models most accurate, though many different distance functions have been proposed in the literature [7].

In this work, the diagonally weighted Euclidean distance,

$$\mathrm{D}_{\mathrm{WE}}(q, x_i) = \left(\sum_{j=1}^{n_d} \lambda_j^2 (q_j - x_{i,j})^2 \right)^{1/2}, \tag{2.2}$$

is used. This metric is much more flexible than the Euclidean distance, but it requires the user to specify the vector of metric weights, $\lambda \in \mathbb{R}^{n_d}$. Chapter 5 describes a new method of optimizing these parameters to maximize model accuracy. The Euclidean metric (Equation 2.1) was used for the examples in this introductory chapter.

2.3 Local Model Types

In principle, any type of local model could be constructed once the k nearest neighbors have been found. The problem, building a nonlinear model given k data points, is in the same form as the larger problem of building a nonlinear model given all n_p data points, where $n_p \gg k$. However, the local modeling problem is unusual for at least two reasons. First, the number of data points is very small and may even be smaller than the number of observed variables, n_d. Most nonlinear modeling methods are not viable under these conditions because they require that the number of data points be at least as large as the dimension of the input vector and usually much larger.

Second, the process of building the model must be repeated for every query. Many popular nonlinear models can not be used locally because too much time is required to construct and evaluate the nonlinear model. For example, although perceptron neural networks can be evaluated quickly they usually require a great deal of initial training data. In the local modeling context this would have to be repeated for each query and would exceed practical time limits for most applications.

2.3.1 Local Linear Models

For the reasons described in the previous section, users prefer models that can be constructed and evaluated very quickly. The two most common types of local models are local averaging models and local linear models; perhaps the two simplest types of models.

Local averaging and local linear models can be described using the same mathematical framework. To see this first consider the local linear model output for an input vector q, which is described by

$$\begin{aligned} \hat{y}(q) &= [q^{\mathrm{T}} \;\; 1]\, \nu, \\ &= \nu_{n_d+1} + \sum_{i=1}^{n_d} q_i \nu_i \,, \end{aligned}$$

where q is the query vector that contains the observed process variables and $\nu \in \mathbb{R}^{n_d+1}$ is a vector of the model coefficients.

The value of the coefficients is found by minimizing the sum of squared errors,

$$\begin{aligned} \mathrm{P}(\nu) &\triangleq \sum_{i=1}^{k} (y_i - \hat{y}_i)^2, \\ &= (b - A\nu)^{\mathrm{T}}(b - A\nu), \\ &= b^{\mathrm{T}}b - \nu^{\mathrm{T}}A^{\mathrm{T}}b - b^{\mathrm{T}}A\nu + \nu^{\mathrm{T}}A^{\mathrm{T}}A\nu. \end{aligned}$$

where $A \in \mathbb{R}^{k \times n_d+1}$ is defined as a matrix that contains each of the k nearest neighboring vectors, each appended with a constant value of one[3], and $b \in \mathbb{R}^k$ is defined as a vector that contains the target outputs for each of the k neighboring points,

$$A \triangleq \begin{bmatrix} x_1^{\mathrm{T}} & 1 \\ x_2^{\mathrm{T}} & 1 \\ \vdots & \\ x_k^{\mathrm{T}} & 1 \end{bmatrix}, \qquad b \triangleq \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix},$$

where x_i is the ith nearest neighbor in the data set[4] and y_i is the target value for x_i.

Since the sum of squared errors is quadratic in ν, there can only be one extremum and it is found by solving for the value of ν that makes the gradient of $\mathrm{P}(\nu)$ equal zero. The gradient is given by

$$\nabla_\nu \mathrm{P}(\nu) = -2A^{\mathrm{T}}b + 2A^{\mathrm{T}}A\nu.$$

Setting this equation equal to zero and solving for ν yields

$$\begin{aligned} \nu &= (A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}b, \\ &= A^{\dagger}b, \end{aligned} \tag{2.3}$$

where $A^{\dagger}$ is the (Moore-Penrose) pseudoinverse of A given by

$$A^{\dagger} \triangleq (A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}.$$

[3]The actual value of this constant is not important. Any constant will result in the same model, though a judicious choice will minimize numeric problems. One is used here for ease of presentation.

[4]The notation is changed slightly from the preceding chapter to simplify notation. Here the subscript for the vectors x_i and the target values y_i is used to denote the ith nearest neighbor instead of the ith point in the data set.

It is not difficult to show that the extremum found by solving $\nabla \mathrm{P}(\nu) = 0$ is the global minimum[5], rather than the global maximum. Since the Hessian of $\mathrm{P}(\nu)$,

$$\nabla_\nu^2 \mathrm{P}(\nu) = 2A^{\mathrm{T}}A,$$

is positive definite for any matrix A with independent columns [42, p. 333], $\mathrm{P}(\nu)$ is a strictly convex function [43, Theorem 3.3.8]. Therefore, the value for ν given in Equation 2.3 globally minimizes $\mathrm{P}(\nu)$ [43, Theorem 3.4.3, Corollary 1].

2.3.2 Local Averaging Models

The framework developed in the previous section can also be used to describe local averaging models. This is done by eliminating the input vectors from the A matrix. In this case, ν is given by

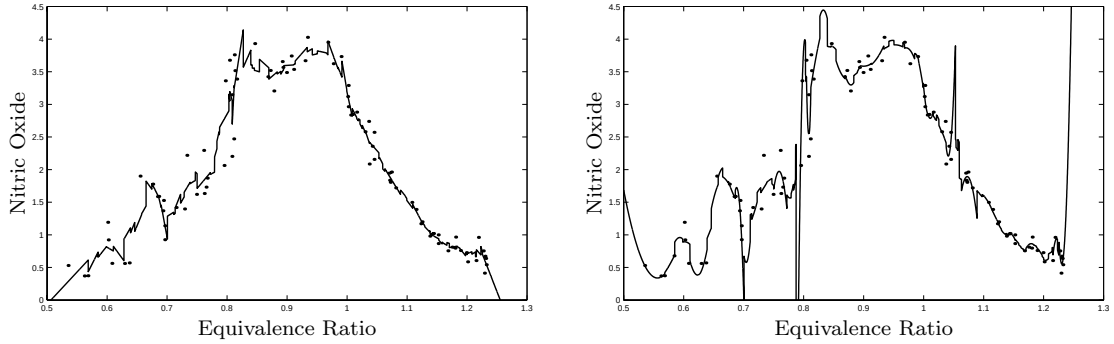
$$\begin{aligned}
\nu &= (A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}b, \\
&= (\mathbf{1}_k^{\mathrm{T}}\mathbf{1}_k)^{-1}\mathbf{1}_k^{\mathrm{T}}b, \\
&= \frac{1}{k}\sum_{i=1}^{k} b_i, \\
&= \frac{1}{k}\sum_{i=1}^{k} y_i, \\
&= \hat{y},
\end{aligned}$$

where $\mathbf{1}_k$ is a k-dimensional vector of ones. In this case, the estimated output, $\hat{y}$, is equal to the sample average of the neighboring target variables.

2.3.3 Local Quadratic Models

Some researchers have also used local quadratic models, but there are many disadvantages of this type of model so it is usually avoided. One disadvantage is that the number of parameters increases quadratically with the number of observed variables, $O(n_d^2)$. Since there must be at least as many points to build the model, the neighborhood size must also scale as $O(n_d^2)$ [34, 35]. Another disadvantage is that quadratic models are very poor at extrapolation and perform poorly in regions where there are few points. For example, Figure 2.2*b* shows the result of using a local quadratic model for the ethanol data set using

[5] If the columns of A are collinear, the global minimum will not be unique.



(*a*) Local Linear (*b*) Local Quadratic

Figure 2.2: Local linear and local quadratic models applied to the Ethanol data set. The number of neighbors was fixed at $k = 5$.

five nearest neighbors. Note that the performance is much worse than that of a local linear model, as shown in Figure 2.2*a*, and that of a local averaging model, as shown in Figure 2.1*b*.

2.3.4 Local Averaging versus Local Linear

Local averaging models and local linear models each have their strengths and weaknesses; neither is universally better. The following sections describe some of the tradeoffs between these two popular types of local models.

Computational Complexity

The computational cost of local averaging scales as $O(k)$ where k is the number of neighbors used to construct the model. The cost of local linear models scales as $O(kn_d^2) + O(n_d^3)$, which is considerably greater than local averaging. However, since k is generally small and n_d is no greater than k, both types of models can be constructed efficiently. Often the computational cost of finding the k nearest neighbors is much greater than the cost of constructing the model, especially for large data sets.

Sensitivity to Regularization

Regularization is a means of compensating for ill-conditioning in the matrix product $A^{\mathrm{T}}A$. This occurs when the columns of A are nearly collinear, as often happens when the number of neighbors is nearly equal to the number of local inputs. In most circumstances the estimate $\hat{y}$ is very sensitive to the method of regularization [44].

This problem does not occur with local averaging models, which have no local inputs and do not require any form of regularization. This also gives local average models the ability to use a single neighbor.

Regularization is discussed more fully in Chapter 4.

Stability

Since the average of a set of numbers can not be greater than the largest number in the set and it can not be smaller than the smallest number in the set, the outputs of local averaging models are guaranteed to be bounded by the range of the nearest neighbors' target values.

Local linear models have no such assurance and can generate outputs that are far outside the range of the data set. This usually occurs when the neighbors in the data set are far away from the query point, which happens frequently in high dimensions, or when the neighbors are collinear. These problems can be mostly alleviated by using an effective method of regularization, but this does not eliminate the problem altogether.

Accuracy

Local linear models generally produce more accurate predictions, especially when an effective method of regularization is used and the data set contains little noise. If the data set is sparsely distributed or contains a lot of noise, local averaging models are sometimes more accurate.

Number of User-Selected Parameters

Local averaging models only require the user to select the number of neighbors, k, and the distance metric. Local linear models require the user to make several additional choices including the method of regularization, the regularization parameters, and which local inputs to use.

2.4 Local Weighting for Smoother Models

Local models are often criticized because their outputs are not smooth functions of the input vector q [18, 36, 45]. Examples of this are illustrated by Figure 2.1b for local averaging and by Figure 2.2a for local linear models. This problem is due to the fact that an arbitrarily small change in the query vector can cause a change in the kth nearest neighbor.

This section describes how to make the output of local averaging models smooth functions of their input variables. Here the term *smooth* means the partial derivative of the

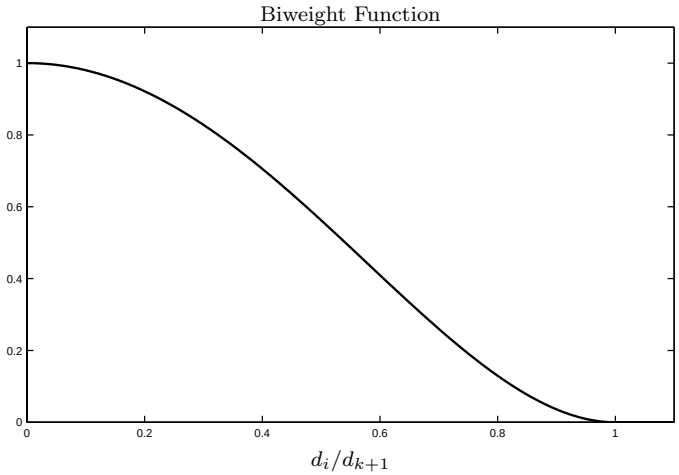


Figure 2.3: The biweight function.

model output with respect to each model input is a continuous function. Section 2.4.2 generalizes this technique to linear models.

2.4.1 Weighted Local Averaging

Local modeling, as described in the previous section, is discontinuous. This problem can be remedied by using an appropriate weighting function that decreases the influence of the furthest neighbor. For example, consider the weighted average

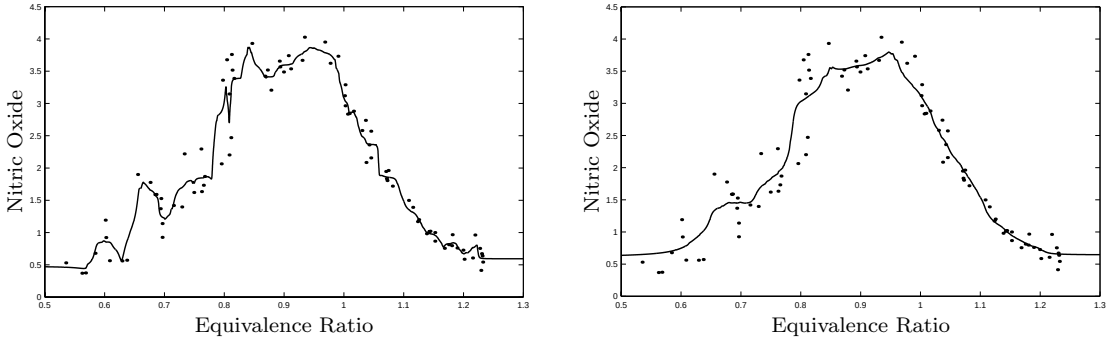
$$\hat{y}(q) = \frac{\sum_{i=1}^{k} y_i w_i^2}{\sum_{i=1}^{k} w_i^2},$$

where each weight w_i^2 is a function of the distance to the ith nearest neighbor[6],

$$w_i^2 = \mathrm{W}\left(\mathrm{D_E}\left(q, x_i\right)\right).$$

The smoothness of $\hat{y}(q)$ depends only on the smoothness of $\mathrm{W}(\cdot)$ and $\mathrm{D_E}(\cdot,\cdot)$. Since the Euclidean distance, $\mathrm{D_E}(\cdot,\cdot)$, is a smooth function, the smoothness of $\hat{y}(q)$ depends only on the weighting function, $\mathrm{W}(\cdot)$.

[6]The reason for using squared weights, w_i^2, is for equivalence with local linear models. This is explained further in the next section.



(*a*) $k = 5$ (*b*) $k = 12$

Figure 2.4: Illustration of local averaging applied to the Ethanol data set.

The accuracy of local models is insensitive to the type of weighting function so long as it is a non-negative, monotonically decreasing, smooth function [7]. A good choice is the biweight function,

$$\mathrm{BW}(d_i, d_{k+1}) \triangleq \begin{cases} \left(1 - \left(\frac{d_i^2}{d_{k+1}^2}\right)\right)^2 & d_i \leq d_{k+1}, \\ 0 & \text{otherwise}, \end{cases} \tag{2.4}$$

where d_i is the distance to the ith nearest neighbor,

$$d_i \triangleq \mathrm{D_E}\,(q, x_i).$$

The biweight function is smooth in the sense that it has a continuous derivative, as illustrated in Figure 2.3.

The biweight function depends only on the ratio of the distance of the ith neighbor to the distance of the $(k+1)$th neighbor. This makes the function invariant to the scale of the metric, $\mathrm{BW}(d_i, d_{k+1}) = \mathrm{BW}(\alpha d_i, \alpha d_{k+1})$ for all $\alpha > 0$. This property will be used in subsequent chapters.

A smooth weighting function guarantees that $\hat{y}(q)$ is smooth except in the case where all $k+1$ of the nearest neighbors are equidistant. This could happen if the $k+1$ neighbors were located on a hypersphere and the query vector were located at the center of the sphere. In this case the sum in the numerator of Equation 2.4.1 would be equal to zero and the value for $\hat{y}(q)$ at this point would depend on the implementation. This is a pathological case, however, and does not occur in practice.

Figure 2.4 shows the result of using weighted local averaging (Equation 2.4.1) on the Ethanol data set for $k = 5$ and $k = 12$ for $w_i^2 = \mathrm{BW}(d_i, d_{k+1})$. Both figures should be compared to Figure 2.1*b* which shows local averaging applied to the same data set with $k = 5$. Although the weighted average results in a continuous model output, it also overfits the data set more than the unweighted average. This is because the closest of the k neighbors have more influence in a weighted average and it is necessary to use a larger number of neighbors to obtain the same degree of accuracy, as shown by Figure 2.4*b*.

2.4.2 Weighted Local Linear Models

The method of local weighting discussed in the previous section can be generalized to the framework of linear modeling described in Section 2.3.1. Consider the linear model that minimizes the weighted sum of squared errors,

$$\begin{aligned} \mathrm{P_W}(\nu) &\triangleq \sum_{i=1}^{k} w_i^2 \left(y_i - \hat{y}_i\right)^2, \\ &= (b - A\nu)^{\mathrm{T}} W^{\mathrm{T}} W (b - A\nu), \\ &= b^{\mathrm{T}} W^{\mathrm{T}} W b - \nu^{\mathrm{T}} A^{\mathrm{T}} W^{\mathrm{T}} W b - b^{\mathrm{T}} W^{\mathrm{T}} W A \nu + \nu^{\mathrm{T}} A^{\mathrm{T}} W^{\mathrm{T}} W A \nu, \end{aligned} \tag{2.5}$$

where $w \in \mathbb{R}^{k,1}$, and $W \in \mathbb{R}^{k \times k}$ is the diagonal matrix of w,

$$\begin{aligned} W &= \mathrm{diag}(w), \\ &= \begin{bmatrix} w_1 & 0 & \cdots & 0 \\ 0 & w_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & w_k \end{bmatrix}. \end{aligned}$$

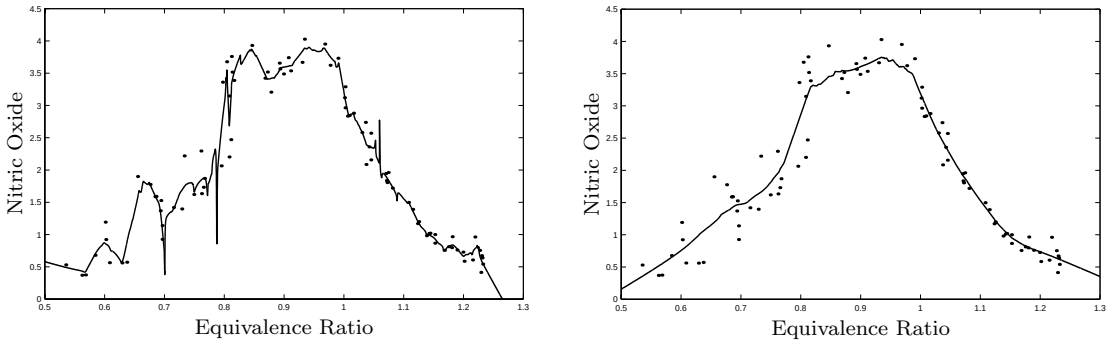
Since W is symmetric, that is $W^{\mathrm{T}} = W$, the product $W^{\mathrm{T}} W$ is subsequently denoted by W^2.

The vector of linear model coefficients that minimizes the weighted sum of squares can be found by using a weighted version of the A matrix and b vector,

$$\begin{aligned} A_W &\triangleq WA, \\ b_W &\triangleq Wb. \end{aligned}$$

After rewriting Equation 2.5 using these new variables,

$$\mathrm{P_W}(v) = b_W^{\mathrm{T}} b_W - v^{\mathrm{T}} A_W^{\mathrm{T}} b_W - b_W^{\mathrm{T}} A_W v + v^{\mathrm{T}} A_W^{\mathrm{T}} A_W v,$$



(*a*) $k = 5$ (*b*) $k = 15$

Figure 2.5: Weighted local linear modeling applied to the Ethanol data set.

it becomes apparent that the weighted sum of squares problem is equivalent to the unweighted problem solved in Section 2.3.1. The solution for the minimizing linear coefficients immediately follows from Equation 2.3,

$$\begin{aligned} v &= (A_W^{\mathrm{T}} A_W)^{-1} A_W^{\mathrm{T}} b_W, \\ &= \left(A^{\mathrm{T}} W^2 A\right)^{-1} A^{\mathrm{T}} W^2 b. \end{aligned} \tag{2.6}$$

Following the same reasoning as in Section 2.3.1, if the query vectors are removed from the A matrix, or equivalently $A = \mathbf{1}_k$, then Equation 2.6 is given by

$$\begin{aligned} v &= \left(\mathbf{1}_k^{\mathrm{T}} W^2 \mathbf{1}_k\right)^{-1} \mathbf{1}_k^{\mathrm{T}} W^2 b, \\ &= (w^{\mathrm{T}} w)^{-1} w^{\mathrm{T}} W b, \\ &= \frac{\sum_{i=1}^k w_i^2 b_i}{\sum_{i=1}^k w_i^2}, \\ &= \frac{\sum_{i=1}^k w_i^2 y_i}{\sum_{i=1}^k w_i^2}. \end{aligned}$$

This is equivalent to the weighted local average described in the previous section (Equation 2.4.1).

Figure 2.5 shows the output of a weighted local linear model applied to the Ethanol data set for $k = 5$ and $k = 15$. As occurs with local averaging, weighting decreases the influence of distant neighbors and causes the model to fit the data set more closely than an unweighted model with the same value for k. Although the output of the weighted model for $k = 5$ shown in Figure 2.5*a* is smooth in that it has a continuous first derivative, the

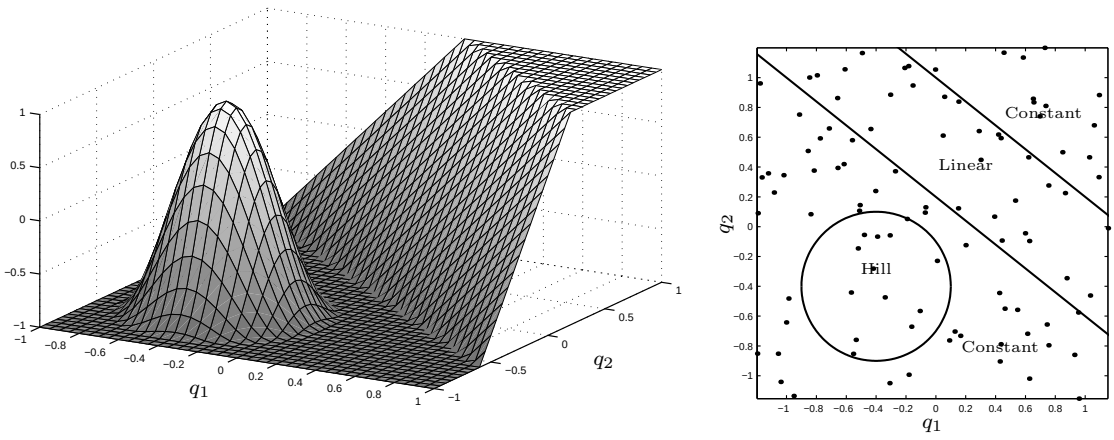
unweighted model output shown in Figure 2.2a is a better fit to the data. If the number of neighbors is increased, the weighted local linear model fits the data more accurately without sacrificing smoothness, as shown in Figure 2.5b.

2.5 The RampHill Data Set

This section illustrates the properties of local modeling applied to a data set with two input variables. This process is a mathematical function that has been used elsewhere to illustrate the advantages and disadvantages of various nonlinear modeling techniques [46, p. 150]. This function is defined as

$$
\begin{aligned}
d_b &\triangleq \sqrt{(q_1+0.4)^2+(q_2+0.4)^2}, \\
y_b &\triangleq \begin{cases} \cos(2\pi d_b) & \text{if } d_b \le 1, \\ 0 & \text{otherwise,} \end{cases} \\
y_\ell &\triangleq 2q_1 + 2.5q_2 - 0.5, \\
y &\triangleq \begin{cases} y_b - 1 & \text{if } y_\ell < 0, \\ y_b + y_\ell - 1 & \text{if } 0 \le y_\ell \le 2, \\ y_b + 1 & \text{otherwise,} \end{cases}
\end{aligned}
$$

where q_1 and q_2 are the function inputs that serve as the observed process variables and y is the output. This function is shown in Figure 2.6. The data set contained 100 points.



(a) (b)

Figure 2.6: (a) Surface plot of the RampHill function. (b) Bird's-eye view of the RampHill function including labels of the significant features and a scatter plot of the data set.

This function was carefully chosen because it contains several features that occur frequently in real processes. First, this function contains two regions in which the output is constant: one at the base of the *hill* and one at the top of the *ramp*. In these regions a nonlinear model should be locally insensitive to the input values.

A second feature of this function is the hill-shaped bump at the left side of the figure. This feature is local in that it only occupies a portion of the input domain. Nonlinear models that are composed of many local features, such as radial basis functions and kernel methods, generally perform well at modeling this type of feature whereas models composed of global features, such as perceptron neural networks and projection pursuit, perform worse.

A third feature of this function is the linear region, or ramp. This is a global feature in that it cuts through the entire input domain. For this type of feature nonlinear models composed of global features, such as perceptron neural networks, generally perform better than nonlinear models composed of local features.

Finally, the RampHill function contains sharp edges at the base of the ramp and at the top of the ramp. Many types of nonlinear models have difficulty with this type of sharp transition because the models are constrained to produce functions that are smooth.

2.5.1 Local Averaging Models

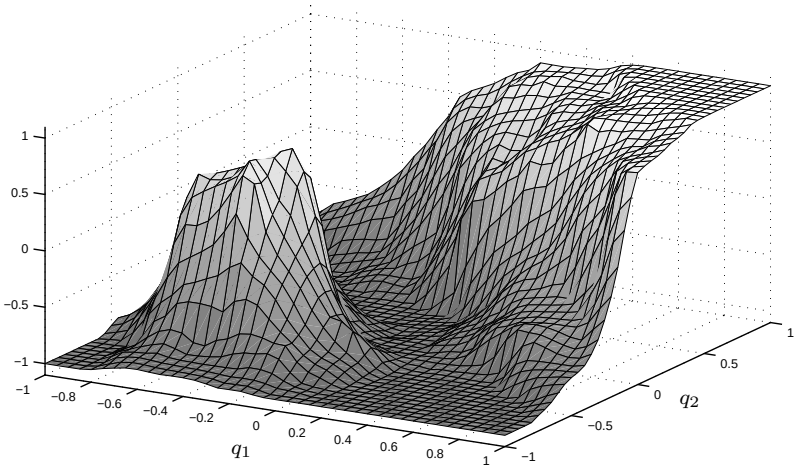
Figure 2.7 shows the result of applying local averaging models to the RampHill data set.

In both cases, the local averaging model approximated the flat region and the hill very well. The approximation of the linear region was smooth, but nonlinear. The model with $k = 5$ had coarser features, but the flat regions were approximated better.

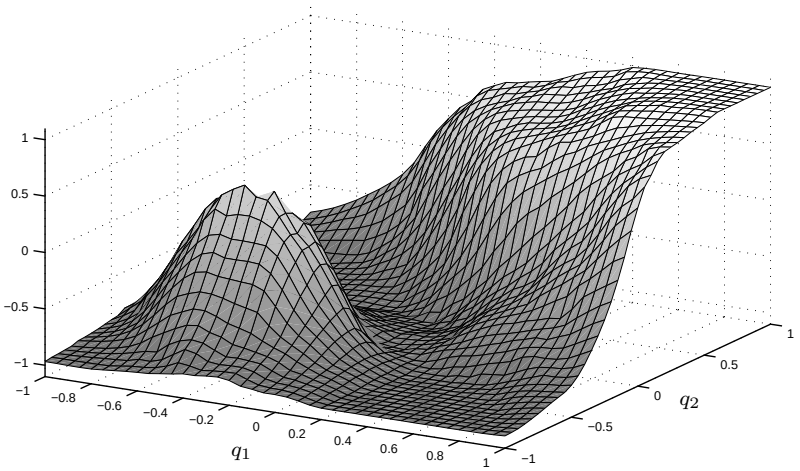
2.5.2 Local Linear Models

Figure 2.8 shows the result of applying local linear models to the RampHill data set.

In both cases the local linear model approximated the linear region much better than the local averaging model. For $k = 5$ the local linear model was much coarser and contained some unstable regions such as the spike to the left of the hill and the overshoot at the top of the ramp. These problems could have been reduced by using an effective method of regularization (see Chapter 4). These problems could also have been reduced by increasing the size of the neighborhood, as illustrated by Figure 2.8*b*. However, this has the disadvantage of over-smoothing the peak of the hill.

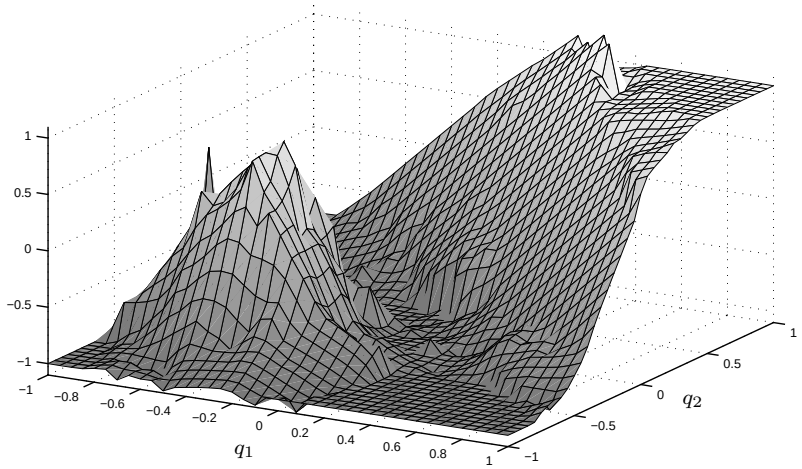


(*a*) $k = 5$

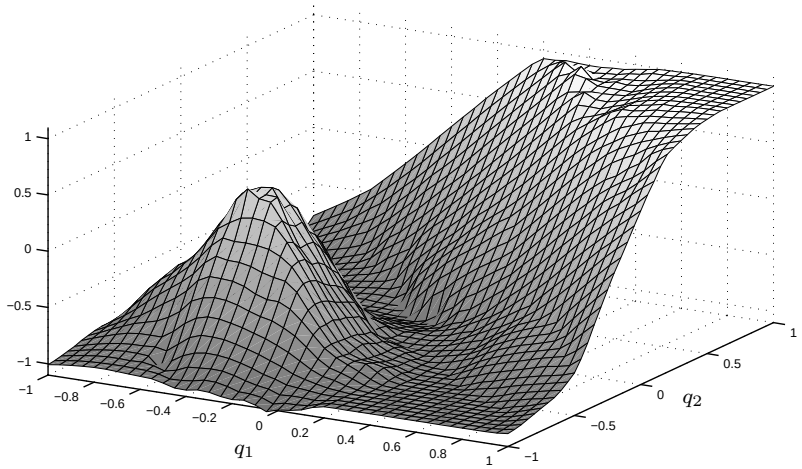


(*b*) $k = 10$

Figure 2.7: The output of a local averaging model for two different values of k.



(*a*) $k = 5$



(*b*) $k = 10$

Figure 2.8: The output of a local linear model for two different values of k.

2.6 The Curse of Dimensionality

The two example data sets used in this chapter do not illustrate how difficult the nonlinear modeling problem is; real processes typically depend on many more than two variables. As the number of dependent variables, called the dimension of the problem, is increased, the process becomes increasingly difficult to model accurately.

The problem is difficult because it is hard to *densely* populate high dimensional spaces with data; this is often referred to as the *curse-of-dimensionality*. For example, if a researcher were to collect data in a controlled experiment and wished to have the data lie on a hyper-grid with ten evenly spaced samples along each coordinate axis, 10^{n_d} points would have to be collected, where n_d denotes the number of dependent variables. In most situations, this quickly becomes impractical for even a modest value of n_d.

As a second example, suppose a data set had an input domain that was limited to a unit hyper-sphere and that the points were arranged so that the distance to the nearest neighbor of each point was equal to a constant, r. An upper bound on the number of points that could exist in this arrangement is given by

$$\begin{aligned} n_p &\leq \frac{A(n_d)}{A(n_d)r^{n_d}}, \\ &= r^{-n_d}, \end{aligned}$$

where

$$A(n_d) = \frac{\pi^{\frac{n_d}{2}}}{\frac{n_d}{2}!},$$

and the volume of an n_d-dimensional hypersphere is given by $A(n_d)r^{n_d}$. This means that the number of points must grow as $O(\left(\frac{1}{r}\right)^{n_d})$ in order to keep the distance to the nearest neighbor constant[7]. As in the previous example, this quickly becomes impractical. Experimental conditions can rarely be controlled to guarantee distributions as uniform as in these two examples and, to compensate, even larger data sets must be collected.

The curse of dimensionality suggests that as the number of observed variables is increased, it becomes more difficult to build an accurate model. This could lead one to falsely conclude that a model that uses only a few of the observed variables will be more accurate than a model that uses all of the observed variables. This conclusion is counterintuitive

[7]This is sometimes called the sphere packing problem.

because a model that only uses a few of the observed variables neglects the information contained in the other variables. This apparent paradox can be resolved by recognizing that neglecting some of the observed variables increases the effective uncertainty in the process output and thereby decreases the model accuracy.

Fortunately, the curse does not necessarily imply that all nonlinear modeling problems with a lot of observed variables are difficult to model accurately; the input space does not always have to be densely populated to construct an accurate model of the process[8]. For example, if only a few of the observed variables are related to the process output, the distribution of the other observed variables is irrelevant.

It is also possible to build accurate models if the domain of the observed variables is limited to a small subspace or nonlinear manifold. This often happens with chaotic systems and is discussed in more detail in Chapters 3 and 6.

2.6.1 The Curse and Local Modeling

For local models a large number of observed variables causes several practical problems. If some of the observed variables are statistically unrelated to the process output, they effectively add noise to the metric. Suppose the process output depends on the first three observed variables and is independent of the remaining variables. The best (Euclidean) distance metric to use in this case is

$$\mathrm{D}_{\mathrm{E}}^{\mathrm{opt}}(q, x_i)^2 = \sum_{j=1}^{3} (q_j - x_{i,j})^2 ,$$

where q is a query vector and x_i is a point in the data set. However, if the Euclidean distance is blindly applied using all of the observed variables, the additional variables $x_{i,4}, \ldots, x_{i,n_d}$ effectively add noise to the optimal metric, as shown by the following:

$$\begin{aligned} \mathrm{D}_{\mathrm{E}}\,(q, x_i)^2 &= \sum_{j=1}^{n_d} (q_j - x_{i,j})^2 , \\ &= \mathrm{D}_{\mathrm{E}}^{\mathrm{opt}}(q, x_i)^2 + \sum_{j=4}^{n_d} (q_j - x_{i,j})^2 , \end{aligned}$$

where the second term represents the noise.

[8] This is discussed thoroughly by Vapnik [47].

Unrelated observed variables also decrease the accuracy of local linear models. This is because the linear model coefficients of the unrelated observed variables will only be approximately zero, since the coefficients are estimated from a very small sample. Extra variables also increase the minimum neighborhood size since the neighborhood must contain at least as many points as there are model coefficients.

In Chapter 5 a new technique is described that optimizes the metric weights and linear model coefficients to diminish the effect of irrelevant model inputs.

2.7 Global Modeling versus Local Modeling

This section gives a brief list of the tradeoffs between local models and global models.

2.7.1 Control of Nonlinearity

One advantage of local models is that they provide direct control of the amount of nonlinearity through the parameter k, the number of neighbors used to construct the model. As k is increased the local model becomes less local and more smooth. In the limit where k becomes equal to the number of points in the data set the local model becomes a global model.

This feature of local models has been used in some instances as a measure of nonlinearity. Casdagli proposed evaluating the performance of local models for many values of k and defined the value of k with the best performance as a measure the process nonlinearity [48].

Some global models also have a single parameter that controls the nonlinearity. For example, radial basis functions have a single smoothing parameter that controls the width of each basis function. However, most global models do not have this property and do not provide information about the amount of process nonlinearity.

2.7.2 Adaptive Complexity

Another advantage of local models is that the degree of nonlinearity adapts to the data set distribution. In regions where the points are spaced far apart the region covered by the nearest neighbors is automatically increased. This efficient use of data is not possible with most global models.

2.7.3 Preprocessing

Since the local model is not constructed until the query vector is available, local models have virtually no preprocessing requirements. However, more advanced local modeling techniques, such as those discussed in subsequent chapters, require some preprocessing to select the local model parameters. If a fast nearest neighbor algorithm is used, it will also probably require some preprocessing.

2.7.4 Computational Cost to Calculate Outputs

Global models can usually generate outputs much more quickly than local models because local models have to find the nearest neighbors and construct the local model for each query.

Fortunately, the time required to generate local model outputs is not impractical. This is mostly because fast algorithms can drastically reduce the time required to find the nearest neighbors, as described in Chapter 3.

The time required to construct the local model is typically less than the time required to find the nearest neighbors. This is true even if a fast nearest neighbor algorithm is used because the local models are built using only a small fraction of the data set and because the local model construction is not iterative.

Since the data set must be searched to find the nearest neighbors, the time required to generate local model outputs grows as the data set size increases. For most data sets taken from real processes, the increase in query time grows as $O(\log n_p)$ where n_p is the number of points in the data set (see Chapter 3 for a detailed discussion). This substantially reduces the cost of using local models on large data sets and makes them viable for most applications.

2.7.5 Limited Selection of Error Measures

A disadvantage of local models is that the optimization of the neighboring points is constrained to minimizing the sum of squared errors, or equivalently, the mean squared error. This is because there is no closed-form solution for the optimal linear model for other error measures. There is some additional flexibility if the weighted sum of squared errors is used, but the flexibility is limited.

Global models usually use an iterative optimization algorithm, sometimes called nonlinear programming, to adapt the model parameters. Most of these algorithms require the gradient of the error with respect to the model parameters to be continuous, which in turn

requires the measure of error to be continuous. This family of error functions includes the weighted sum of squared errors used by local models and many other error functions.

The iterative optimization used by global models could also be used to minimize a different error measure for local models, but iterative optimization is much slower than calculating the optimal least squares solution. Since the optimization must be repeated for each query, the additional cost is usually prohibitive.

2.7.6 Efficient Calculation of Cross-Validation Error

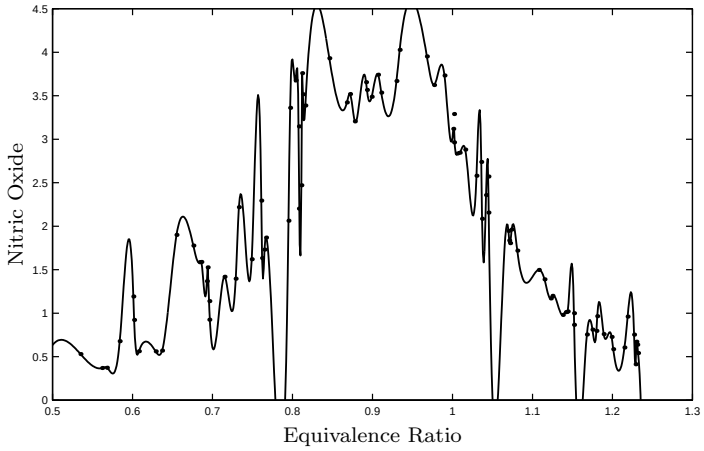


Figure 2.9: Illustration of overfitting. This shows the output of a cubic spline constructed using the Ethanol data set.

Almost all nonlinear models optimize model parameters to minimize some measure of performance. In most cases the measure of performance is a measure of average error, such as mean squared error, taken over the entire data set. This approach often causes the model to be accurate at the points in the data set but to vary substantially at other points; this effect is called overfitting. For example, Figure 2.9 shows a cubic spline overfitting the Ethanol data set.

To solve this problem, users will often divide the data set into two parts: a training data set and a test data set. The nonlinear model is then iteratively optimized to minimize the average error on the training data set and the optimization is stopped once the average error on the test set increases.

Local models can use a much more accurate technique of estimating the model performance. This technique consists of taking a single point out of the data set, building a nonlinear model using the remaining $n_p - 1$ points, and using the nonlinear model to estimate the output at the removed point. The process is repeated for all n_p points in the data

set and the average error is calculated. This error is called the leave-one-out cross-validation error (CVE).

The computational cost of calculating the CVE is prohibitive for most global models because it requires the model to be constructed n_p times. Calculating the CVE multiple times, as would be necessary to use the CVE in an iterative optimization of model parameters, is even more daunting.

Local models can calculate the CVE almost as efficiently as they can calculate the local model outputs. The number of nearest neighbors is increased by one and after the $k+2$ nearest neighbors have been found, the nearest neighbor, which is identical to the query vector, is discarded and the model is constructed using the remaining $k+1$ neighbors.

The ability to calculate the cross-validation error efficiently is a very important advantage of local models and it plays a vital role in the optimization algorithms described in Chapters 5 and 6.

2.8 Summary

This chapter laid the mathematical foundation for local modeling that is built upon in subsequent chapters. Some of the disadvantages of local modeling were also described in this chapter. One of these disadvantages, that local models are discontinuous, was circumvented by the use of local weighting.

Other problems introduced in this chapter are addressed in the following chapters. The next chapter addresses the large amount of computation required to find the nearest neighbors for each query. This problem is solved by using fast nearest neighbor algorithms.

Another problem introduced in this chapter is the inherent instability of local linear models. Chapter 4 describes two new methods of regularization that address this problem.

This chapter also described the curse of dimensionality and the problems caused by building local models with input variables that are unrelated to the process output. These problems are addressed in Chapter 5, which introduces a method for adapting the regularization and distance metric to diminish the effect of irrelevant input variables.

Finally, this chapter briefly described some of the tradeoffs between local models and global models. The most important advantage of local models is their ability to accurately and efficiently measure model performance through the use of leave-one-out cross-validation. This property enables the model parameters to be optimized without the danger of overfitting.

Chapter 3

Fast Nearest Neighbor Algorithms

640K ought to be enough for anybody.
—Bill Gates, 1981

One of the disadvantages of local modeling arises from the large amount of computation required to find the nearest neighbors—especially for large data sets. Fortunately, fast nearest neighbor algorithms can drastically diminish the computation required to find the nearest neighbors. This chapter introduces two new fast algorithms and describes a thorough comparative study of these and other leading nearest neighbor algorithms.

The following section reviews the most frequent techniques that fast algorithms use to reduce query time. Section 3.3 discusses how the dimension of the data set affects these methods. Section 3.4 describes four nearest neighbor algorithms in detail, including the two new algorithms. Section 3.5 specifies the other algorithms included in this study, discusses different measures of query time, and describes how the user-specified parameters were chosen. Section 3.6 reports the performance of the algorithms on a variety of benchmark problems.

3.1 Introduction

In this chapter two new algorithms are introduced and compared with other leading algorithms. Only algorithms that use the Euclidean metric to measure the distance between two points,

$$\mathrm{D}(x,q) \triangleq \left(\sum_{i=1}^{n_d}(x_i - q_i)^2\right)^{1/2},$$

are included in this chapter. However, these algorithms can also be used with a diagonally weighted Euclidean distance as described in Chapter 1.

Fast algorithms have also been developed for several closely related problems that are not addressed here. For example, fast nearest neighbor algorithms have been developed for non-Euclidean metrics [49–51], for finding only the *nearest* neighbor [52], and for finding only the approximate nearest neighbors [53]. Also, some algorithms are designed specifically for images [54–57] and some approximate algorithms have been developed that reduce computation at the expense of accuracy [13, 51–53, 58–64].

The performance of nearest neighbor algorithms is usually specified by the preprocessing time, memory required, and average query time. For most applications, a moderate amount of preprocessing and memory allocation are acceptable if the reduction in query time is large enough.

3.2 Review of Elimination Criteria

Many nearest neighbor algorithms have been proposed to overcome the large computational cost of the brute force approach of calculating the distance from the query to each point in the data set. Typically, these algorithms employ one or more of the elimination criteria discussed in this section. Each estimates a lower bound on the distance between a query point and a point, or set of points, in the data set. If the lower bound is greater than the distance to the kth nearest neighbor found so far, the point can be eliminated without explicitly calculating the distance to that point[1].

[1] Nene and Nayar give a complementary review of nearest neighbor algorithms developed in computational geometry [62].

3.2.1 Full-Search Improvements

Full-search improvements compute one or more lower bounds for each point in the data set. A common improvement is to check whether the candidate is outside of the hypercube that contains the k nearest neighbors found so far [13, 62, 65–67]. Other methods use each vector's mean [66, 68–70] or a combination of the vector's mean and variance [71, 72].

Torres and Huguet proposed a method that reduces the cost of calculating the Euclidean distance between two points [73]. Cheng *et al.* proposed a similar method that stops the distance calculation if the partial distance exceeds a known threshold.

Several researchers have proposed methods to order the points so as to reduce computation [74–76] and others have proposed ordering the elements of the distance summation for the same purpose [77].

Because full-search algorithms consider each point in the data set explicitly, the query time scales as $O(n_p)$. Although they may be several times faster than the brute force algorithm, for large data sets they are generally slower than $O(\log n_p)$ algorithms, especially in low dimensions.

3.2.2 Search Trees

During preprocessing, search trees divide the data set into distinct subsets. Each subset is recursively subdivided until the number of points in each terminal node is acceptably small. The search for the nearest neighbors begins with the root node and works toward the terminal nodes. At each step, a lower bound on the minimum distance to each subset of points is calculated. If the lower bound is greater than the distance to the k nearest neighbors found so far, the entire subset can be eliminated without calculating the distance to each point explicitly [78–83].

Because search trees are capable of eliminating entire groups of points, query times of $O(\log n_p)$ are frequently achieved in low dimensions [79, 80].

3.2.3 Growing Search Regions

In many cases the points in a specified region of the search space can be found very quickly. Growing region methods start with a region of some user-specified size and check if the k nearest neighbors are contained in this region. If too few points are found, the region size is increased and the process repeated until at least k neighbors are found [12, 13, 58, 62, 84].

A common type of growing region method begins by checking the points that are contained in a small hypercube centered at the query point. If no points are contained in the hypercube, then the side length is increased and the process is repeated.

Growing region methods have two significant disadvantages. First, for data sets with unknown distributions, there is no efficient and accurate technique for estimating the initial size of the region [62]. This problem is exacerbated in high dimensions for reasons discussed in Section 3.3. Second, the difference between the volume of the bounding region and the hypersphere that contains the k nearest neighbors grows exponentially as the dimension increases [33]. For example, in high dimensions most of the points fit in the hypercube that encloses the hypersphere containing only the k nearest neighbors [62].

3.2.4 Axis-Partitioning Algorithms

Axis-partitioning algorithms divide the n_d-dimensional input space into hyperrectangles and calculate a lower bound on the distance from the query point to all points contained in each hyperrectangle. If the lower bound is greater than the distance to the kth nearest neighbor found so far, all of the points contained by the hyperrectangle can be eliminated without explicitly calculating the distance to each point contained therein [13, 78–80, 85–88].

Zakarauskas and Ozard compared four of the leading axis-partitioning algorithms and concluded that they have very similar performance [89].

3.2.5 Triangle Inequality

Frequently the triangle inequality is used to cheaply calculate a lower bound on the distance to a point or set of points [5, 12, 81, 82, 85, 90–103]. Many algorithms begin a search by calculating the distance from the query point to an anchor point, a. During preprocessing, the distance from the point a to each point in the data set is calculated, $\mathrm{D}(a, x_i)$ for all i. The triangle inequality can then be used to establish two lower bounds on the distance from the query point to the point x_i,

$$\mathrm{D}(q, x_i) \geq \mathrm{D}(a, q) - \mathrm{D}(a, x_i),$$

and

$$\mathrm{D}(q, x_i) \geq \mathrm{D}(a, x_i) - \mathrm{D}(a, q).$$

These lower bounds can be combined into a single equation:

$$\mathrm{D}(q, x_i) \geq |\mathrm{D}(a, x_i) - \mathrm{D}(a, q)|. \tag{3.1}$$

If the distance to the kth nearest neighbor found so far is less than $|\mathrm{D}(a, x_i) - \mathrm{D}(a, q)|$, x_i can not be a nearest neighbor of q.

3.2.6 Projection Methods

Projection methods reduce the computation of other elimination criteria by projecting the data set and the query point into a linear subspace, which is usually spanned by the first few principal components [6, 11, 55, 104–107]. However, projections have also been used for other purposes such as choosing the locations of the anchor points for the triangle inequality [12, 95] and ordering the search sequence [76].

In Section 3.4 two new algorithms are described that use the principal component projection to construct efficient search trees.

3.3 The Problem of Dimensionality

Many of the elimination criteria described in the previous section require $O(n_p \log n_p)$ preprocessing time, $O(n_p \log n_p)$ storage space, and achieve $O(\log n_p)$ mean query time or better in low dimensions, where n_p is the number of points in the data set [79, 80, 82]. However, if the data set variables are independently distributed, the elimination criteria become less effective as the dimension increases and the average query times of the fastest algorithms scale exponentially with dimension. When the dimension becomes large enough, the elimination criteria become mostly ineffective and the distances are calculated for nearly all of the points in the data set. At this dimension, the average query time transitions from an exponential growth to linear growth. This is illustrated for several algorithms in Figure 3.4. If the dimension is large enough, the fastest algorithms can have a larger query time than full-search algorithms due to the computational overhead of the elimination criteria.

Much better performance is typically observed in practice because real data sets are rarely composed of independently distributed variables. The intrinsic dimension, defined as the minimum number of parameters needed to account for the observed properties of a data set, is often much less than the spatial dimension [108, 109]. Conceptually, this occurs when all of the points in a data set lie on a n_i-dimensional nonlinear surface, or manifold, in a n_d-dimensional space, where $n_i \ll n_d$.

The elimination criteria discussed in the previous section vary in effectiveness for data sets with small intrinsic dimension. For example, the effectiveness of the triangle inequality does not change as the spatial dimension is increased so long as the interpoint distances remain the same. In this case the query time scales linearly with the spatial dimension, n_d, rather than exponentially. In contrast, axis-partitioning algorithms are generally more sensitive to spatial dimension since the partitions always occur along the coordinate axes; but this is mitigated by the ability of these algorithms to concentrate the partitions in the highest density regions.

The two new algorithms described in the following section use principal component analysis to adapt to the data set distribution and they perform well when the intrinsic dimension is less than the spatial dimension.

3.4 Algorithms

The algorithms described below all find the exact nearest neighbors of a query point. Most of these algorithms also calculate the distances to the nearest neighbors in the process, though some of the algorithms calculate only the squared distances. For this study, the algorithms were only required to find the indices of the neighboring points, not the distances.

All of the algorithms described here assume that the data set consists of n_p points, $\{x_1, x_2, \ldots, x_{n_p}\}$, where each point exists in a n_d-dimensional space, $x_i \in \mathbb{R}^{n_d}$ for $i = 1, 2, \ldots, n_p$. The vector $d \in \mathbb{R}^k$ is used to store the sorted distances to the nearest neighbors found so far and d^2 is used to represent a vector of the square of these distances.

3.4.1 Brute Force (Brute)

The most obvious implementation of a nearest neighbor algorithm is the brute force algorithm, which requires no preprocessing or additional storage. The algorithm is given below. Note that only the squared distances are used, $\mathrm{D}^2(x_i, q)$, to avoid the square root calculations.

Algorithm 3.1: Brute Force Search

1. $d_i^2 := \infty$ for $i = 1, 2, \ldots, k$.
2. For $i = 1$ to n_p,
   - 2.1 Calculate $\mathrm{D}^2(x_i, q)$.
   - 2.2 If $\mathrm{D}^2(x_i, q) < d_k^2$, then update the list of nearest neighbors and d^2.
   - 2.3 Next i.

This algorithm calculates the distance to all points in the data set and in the process maintains a list of the k nearest neighbors found so far.

The average query time of this algorithm provides a performance baseline against which the other algorithms are compared.

3.4.2 Partial Distance Search (PDS)

Cheng *et al.* [13] originally proposed the partial distance search algorithm, but others have also proposed it independently [74,77,110]. The algorithm consists of a simple modification of the brute force search: during the calculation of the distance $\mathrm{D}^2(x_i, q)$, if the partial distance, defined as $\sum_{i=1}^{n_p}(x_{i,j} - q_j)^2$ for $n_p < n_d$, exceeds the distance to the kth nearest neighbor found so far, the calculation is aborted. Like Brute, PDS does not require any preprocessing or storage. The partial distance search algorithm is given below.

Algorithm 3.2: Partial Distance Search

1. Define s as a scalar variable that contains the partial distance.
2. $d_i^2 := \infty$ for $i = 1, 2, \ldots, k$.
3. For $i =$ 1 to n_p,
   - 3.1 $s := 0$.
   - 3.2 For $j =$ 1 to n_d,
     - $s := s + (x_{i,j} - q_j)^2$.
     - If $s > d_k^2$, then x_i is not a nearest neighbor of q. Next i.
     - Next j.
   - 3.3 Update the list of nearest neighbors and d^2.
   - 3.4 Next i.

In almost all cases, aborting the distance calculation early improves the performance of this algorithm as compared to Brute.

3.4.3 Principal Axis Tree (PAT)

This section describes a new method of using principal component analysis (PCA) to build an efficient search tree. Although PCA has been incorporated into other nearest neighbor algorithms, it has not been combined with the efficient structure of a search tree or the elimination criteria described here.

The search tree construction begins by projecting the entire data set along the principal axis, defined as the principal component with the largest eigenvalue. Next, the data set is partitioned along the principal axis into n_c distinct regions such that each region contains

roughly the same number of points. The process is repeated for each subset of points recursively until each subset contains fewer than n_c points. A two-dimensional example of this type of partition is shown in Figure 3.1. The algorithm for the search tree construction is given below. This algorithm requires $O(n_p \log n_p)$ preprocessing and storage.

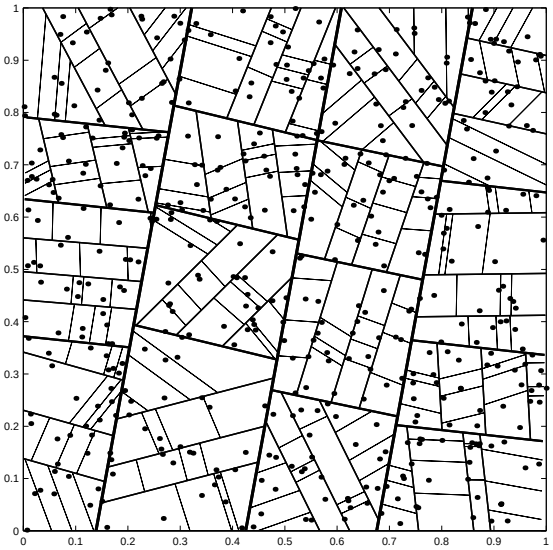


Figure 3.1: Partition of a two-dimensional data set using a principal axis tree. Each non-terminal node in the tree subdivided its assigned set of points into four subsets.

Algorithm 3.3: Principal Axis Tree Construction

1. Define n_z as the number of points assigned to a node N and define $z_i \in \mathbb{R}^{n_d}$ as the ith point. Define n_c as the number of child nodes assigned to each non-terminal node.
2. $d_i^2 := \infty$ for $i = 1, 2, \ldots, k$.
3. $N :=$ the root node.
4. Assign all of the points in the data set to N.
5. BuildTree(N).

Function BuildTree(N)

1. If $n_z < n_c$, then N is a terminal node. Exit function.
2. Calculate the principal axis, p, for the points assigned to N.
3. Calculate the projections of all of the points assigned to N onto the principal axis: $g_i := \sum_{j=1}^{n_d} z_{i,j} p_j$ for $i = 1, 2, \ldots, n_z$.
4. Sort the projected values in increasing order.
5. Divide the principal axis into n_c distinct regions such that the number of projected values in each region is either $\lfloor n_z / n_c \rfloor$ or $\lfloor n_z / n_c \rfloor + 1$. Define G_i as the set of points contained in the ith region for $i = 1, 2, \ldots, n_c$.

6. Calculate the smallest and largest projection for each subset of points, G_i:
   $g_{i,\min} := \min_{z_j \in G_i} g_j$,
   $g_{i,\max} := \max_{z_j \in G_i} g_j$.
7. Create a child node for each subset of points.
   For $i = 1$ to n_c,
   7.1 Create a child node C_i and assign the points in the subset G_i to this node.
   7.2 BuildTree(C_i).
   7.3 Next i.

The principal axis in Step 2 of BuildTree(N) can be efficiently calculated by the power method [111]. This technique has also been used for codebook generation in vector quantization and for clustering algorithms [112]. The implementation of the power method used in PAT is described below. See Golub and Van Loan for a discussion of the convergence properties of this method [111].

Algorithm 3.4: Power Method for Principal Axis Tree

1. Define n_z as the number of points used to find the principal axis and define $z_i \in \mathbb{R}^{n_d}$ as the ith point.
2. Calculate the sample mean:
   $\bar{z} := \frac{1}{n_z} \sum_{i=1}^{n_z} z_i$.
3. Pick the diagonal axis as the initial value for p:
   $p_i := \frac{1}{\sqrt{n_d}}$ for $i = 1, 2, \ldots, n_d$.
4. Initialize the stopping-criteria variables:
   $\sigma^2_{\text{prev}} := 0$,
   $n_\ell := 1$.
5. Project the centered points onto p and store in a vector $u \in \mathbb{R}^{n_z}$.
   For $i = 1$ to n_z,
   5.1 $u_i := \sum_{j=1}^{n_d} (z_{i,j} - \bar{z}_j) p_j$.
   5.2 Next i.
6. Calculate the variance in the direction of the projection p:
   $\sigma^2 := \frac{1}{n_z} \sum_{i=1}^{n_z} u_i^2$.
7. Check convergence criteria:
   If $(\sigma^2 - \sigma^2_{\text{prev}})/\sigma^2 < \epsilon$ and $n_\ell > n_{\min}$, then p has converged to the principal axis. Exit function.
8. Calculate the product of the centered covariance matrix times p and store in a vector $r \in \mathbb{R}^{n_d}$.
   For $i = 1$ to n_d,
   8.1 $r_i := \sum_{j=1}^{n_z} (z_{j,i} - \bar{z}_i) u_j$.
   8.2 Next i.
9. Normalize the vector r and update p:
   $p := r / \sqrt{\sum_{i=1}^{n_d} r_i^2}$.

10. Update the stopping criteria variables:
    $\sigma^2_{\text{prev}} := \sigma^2$ and
    $n_\ell := n_\ell + 1$.
11. Goto 5.

This algorithm has two user-specified parameters, ϵ and $n_{\min}$. The accuracy of p is fairly insensitive to the values of these parameters. For all of the results reported here these parameters were fixed at $\epsilon = 0.001$ and $n_{\min} = 20$.

After the search tree is constructed, the k nearest neighbors can be found using a depth-first search. The process begins with the root node and uses a binary search to determine which region the query point is in. The child node that contains this region is then searched, and the process is repeated recursively until a terminal node is reached. Partial distortion search is used to calculate the distance to points in the terminal node. The remaining children of each ancestor node are searched if an elimination criterion is not satisfied. The process continues until the root node is reached and all of the child nodes have been searched or eliminated. This search process is described in detail by the algorithm given below.

Algorithm 3.5: Principal Axis Tree Search

1. Define N as a node in the tree. If N is a non-terminal node, define n_c as the number of child nodes contained in N, C_i as the ith child node, and G_i as the group of points assigned to C_i. Define p as the projection vector of N and define $g_{i,\max}$ and $g_{i,\min}$ as the maximum and minimum values of the points in G_i projected onto p. If N is a terminal node, define n_z as the number of points assigned to N and $z_i \in \mathbb{R}^{n_d}$ as the ith point.
2. $d_i^2 := \infty$ for $i = 1, 2, \ldots, k$.
3. $N :=$ the root node.
4. Search(N,q,q,0).

Function Search(N,q,b,d^2_{LB})

1. If N is a terminal node, then
   1.1 Perform a Partial Distance Search on the points assigned to this node.
   For $i = 1$ to n_z,
   - $s := 0$.
   - For j = 1 to n_d,
     - $s := s + (z_{i,j} - q_j)^2$.
     - If $s > d_k^2$, then z_i is not a nearest neighbor of q. Next i.
     - Next j.
   - Update the list of nearest neighbors and d^2.
   - Next i.

1.2 Exit function.

2. Calculate the projection of the boundary point onto p:
   $\sigma := \sum_{i=1}^{n_d} b_i p_i$.

3. Initialize stopping-criteria variables:
   LowerDone := false,
   UpperDone := false.

4. Find the closest group of points using the upper boundaries. Begin by checking end conditions.
   If $\sigma < g_{1,\max}$, then $i_u := 1$ and LowerDone := true.
   Else if $\sigma > g_{n_c-1,\max}$, then $i_u := n_c$, $i_\ell := n_c - 1$, and UpperDone := true.
   Else, perform a binary search:

   4.1 $i_\ell := 1$.

   4.2 $i_u := n_c$.

   4.3 While $i_u - i_\ell > 1$,

   · $i := \lfloor (i_u + i_\ell)/2 \rfloor$.

   · If $\sigma < g_{i,\max}$, then $i_u := i$.
   Else, $i_\ell := i$.

5. Search the child node assigned the region that encloses b.
   Search(C_{i_u},q,b,d^2_{LB}) and
   $i_u := i_u + 1$.

6. Initalize variables for main loop:
   If LowerDone = false, then $d_\ell := \sigma - g_{i_\ell,\max}$.
   If UpperDone = false, then $d_u := g_{i_u,\min} - \sigma$.

7. Search the remaining child nodes starting with the closest until the bounding conditions apply.
   While UpperDone = false or LowerDone = false,

   7.1 If (UpperDone = true or $d_\ell < d_u$) and LowerDone = false, then check C_{i_ℓ}.

   · Calculate lower bound on distance to points in G_{i_ℓ}:
   $d^2_{\mathrm{LB}_\ell} := d^2_{\mathrm{LB}} + d^2_\ell$.

   · If $d^2_k < d^2_{\mathrm{LB}_\ell}$, then none of the points in G_i for $i = 1, 2, \ldots, i_\ell$ is a nearest neighbor of q. LowerDone := true. Goto 7.

   · If C_{i_ℓ} is not a terminal node, then calculate the boundary point:
   $b_\ell := b - d_\ell p$.

   · Search(C_{i_ℓ},q,b_ℓ,$d^2_{\mathrm{LB}_\ell}$).

   · $i_\ell := i_\ell - 1$.

   · If $i_\ell = 0$, then LowerDone := true.
   Else, $d_\ell := \sigma - g_{i_\ell,\max}$.

   · Goto 7.

   7.2 Else, check node C_{i_u}

   · Calculate lower bound on distance to points in G_{i_u}:
   $d^2_{\mathrm{LB}_u} := d^2_{\mathrm{LB}} + d^2_u$.

- If $d_k^2 < d_{\mathrm{LB}_u}^2$, then none of the points in G_i for $i = i_u, i_u+1, \ldots, n_c$ is a nearest neighbor of q. UpperDone := true. Goto 7.
- If C_{i_u} is not a terminal node, then calculate the boundary point: $b_u := b + d_u p$.
- Search(C_{i_u},q,b_u,$d_{\mathrm{LB}_u}^2$).
- $i_u := i_u + 1$.
- If $i_u = n_c + 1$, then UpperDone := true. Else, $d_u := g_{i_u,\min} - \sigma$.
- Goto 7.

This algorithm searches the tree depth-first until it locates the terminal node region that encloses the query point. The algorithm then works its way back up to the root node, either searching or eliminating sibling nodes along the way.

Before each sibling node is searched, a lower bound on the distance to the points assigned to the node is calculated. If the lower bound is greater than the distance to the kth nearest neighbor found so far, none of the points assigned to the node is a neighbor and the node does not need to be searched. Other siblings can also be eliminated that have larger lower bounds.

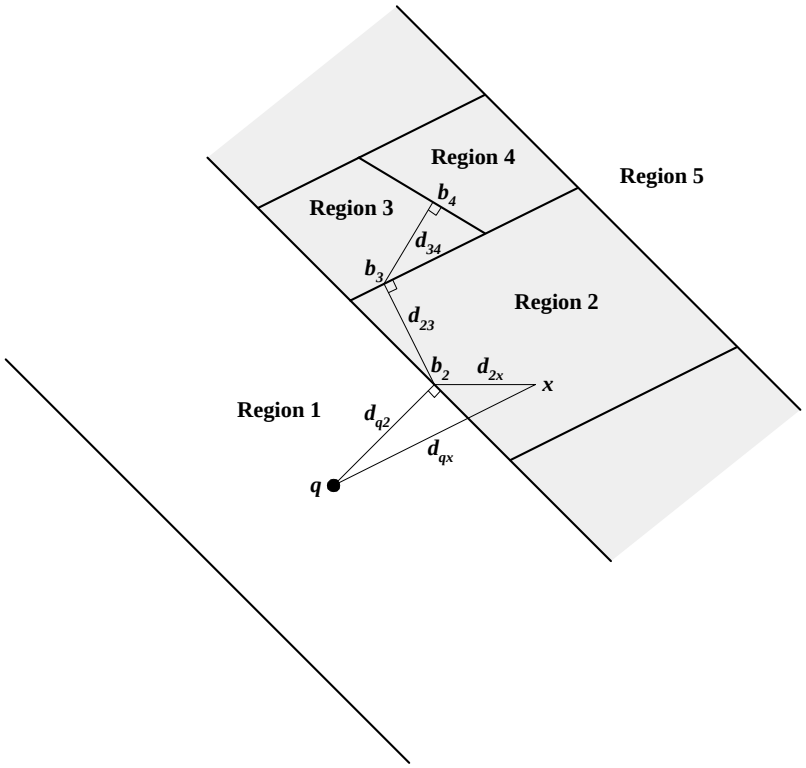


Figure 3.2: Illustration of the bounding criteria of principal axis trees.

The calculation of the lower bound is illustrated in Figure 3.2. During the construction of the search tree, the points assigned to each node are subdivided by hyperplanes located along the principal axis. The distance from the query point, q, to the boundary of a child node, b_2, is shown by the chord labeled d_{q2}. This distance is a lower bound on the distance to all of the points in the gray region. If this distance is greater than the distance to the kth nearest neighbor found so far, then all of the points in this region can be eliminated. Similarly, all of the regions that are further away along the principal axis can be eliminated including, for example, Region 5.

If the lower bound does not apply, a different bound may be used to calculate the minimum distance to a point within the region. For example, consider a point x in Region 2. Since the chord $\overline{qb_2}$ is a normal of the hyperplane separating Region 1 from Region 2, the angle $\measuredangle qb_2x$ is no smaller than 90°. A lower bound on the distance between q and x is given by the law of cosines:

$$\begin{aligned} d_{qx}^2 &= d_{q2}^2 + d_{2x}^2 - 2d_{q2}d_{2x}\cos\left(\measuredangle qb_2x\right), \\ &\geq d_{q2}^2 + d_{2x}^2. \end{aligned} \tag{3.2}$$

This method can also be used to establish a lower bound for subregions of the gray region. For example, if the boundary point between Region 2 and Region 3 is calculated (shown as b_3 in Figure 3.2), a lower bound on the distance from b_2 to the points in Region 3 can be calculated by projecting b_2 onto the principal axis of the gray region and calculating the distance from b_2 to b_3. Since d_{23}^2 is a lower bound on the distance from b_2 to any point in Region 3, Equation 3.2 can be used to find a lower bound on the distance from q to any point in Region 3:

$$\begin{array}{lcll} \mathrm{D}^2(q,x) & \geq & d_{q2}^2 + d_{2x}^2 & \forall x \in \text{ Gray Region}, \\ d_{2x}^2 & \geq & d_{23}^2 & \forall x \in \text{ Region 3, and therefore} \\ \mathrm{D}^2(q,x) & \geq & d_{q2}^2 + d_{23}^2 & \forall x \in \text{ Region 3}. \end{array}$$

The same reasoning can be extended to other subregions. For example, a lower bound on the distance from q to any point in Region 4 is given by

$$\mathrm{D}^2(q,x) \geq d_{q2}^2 + d_{23}^2 + d_{34}^2 \quad \forall x \in \text{ Region 4}. \tag{3.3}$$

The boundary points, b, and distance lower bounds, d_{LB}^2, are calculated recursively in the search algorithm for PAT.

3.4.4 Depth-Only Principal Axis Tree (DOPAT)

A depth-only search similar to that proposed by Katsavounidis *et al.* [83] can also be used with the principal axis search tree. The construction algorithm is the same as for PAT, except the projection of each point onto the principal axis of each non-terminal node, $g_i \ \forall i$, is saved instead of the region boundaries.

The search algorithm begins by finding the terminal node region that encloses the query point, just like PAT. However, DOPAT does not explore any sibling nodes as it works its way back to the root node. Instead, a lower bound is calculated for all of the points assigned to each ancestor of the terminal node until the lower bound exceeds the distance to the nearest neighbor found so far.

This approach could be used for an arbitrary number of partitions, but the query time is almost always minimal when the number of partitions is two. In this case, a more efficient algorithm exists that explicitly incorporates this property, as described below. The generalization to more partitions is straightforward.

Algorithm 3.6: Depth-only Principal Axis Tree Search

1. Define N as a node in the tree. If N is a non-terminal node, define b as the index of the boundary point of the partition,z_b. Define C_ℓ as the child node that is assigned all of the points that have projections less than the boundary point. Define C_r as the child node assigned all of the remaining points and define p as the projection vector of N. Define n_z as the number of points assigned to N and $z_i \in \mathbb{R}^{n_d}$ as the ith point. Define g_i as the value of z_i projected onto p and let the points z_i be sorted in increasing order of their projections, g_i.
2. $d_i^2 := \infty$ for $i = 1, 2, \ldots, k$.
3. $N :=$ the root node.
4. Search(N,q).

Function Search(N,q)

1. If N is a terminal node, then
   - 1.1 Perform a Partial Distance Search on the points assigned to this node.
     For $i = 1$ to n_z,
     - $s := 0$.
     - For $j = 1$ to n_d
       - $s := s + (z_{i,j} - q_j)^2$.
       - If $s > d_k^2$, then z_i is not a nearest neighbor of q. Next i.
       - Next j.
     - Update the list of nearest neighbors and d^2.
     - Next i.
   - 1.2 Exit function.

2. Calculate the projection of the query point onto p:
   $\sigma := \sum_{i=1}^{n_d} q_i p_i$.
3. Calculate the distance to the kth nearest neighbor:
   $d_k := \sqrt{d_k^2}$.
4. If $\sigma < g_b$, then search the left node.
   4.1 Search(C_ℓ,q).
   4.2 Check the points on the right side of the boundary.
   For $i = b$ to n_z,
   - If $d_k < g_i - \sigma$, then all of the remaining points on the right side of the boundary are not a nearest neighbor of q. Exit function.
   - Calculate the partial distance to z_i.
     $s := 0$.
   - For $j = 1$ to n_d,
     - $s := s + (z_{i,j} - q_j)^2$.
     - If $s > d_k^2$, then z_i is not a nearest neighbor of q. Next i.
     - Next j.
   - Update the list of nearest neighbors, d^2, and d_k:
     $d_k := \sqrt{d_k^2}$.
   - Next i.

   4.3 Exit function.
5. Else, search the right node.
   5.1 Search(C_r,q).
   5.2 Check the points on the left side of the boundary.
   For $i = b-1$ to 1 Step -1,
   - If $d_k < \sigma - g_i$, then all of the remaining points on the left side of the boundary are not a nearest neighbor of q. Exit function.
   - Calculate the partial distance to z_i.
     $s := 0$.
   - For $j = 1$ to n_d,
     - $s := s + (z_{i,j} - q_j)^2$.
     - If $s > d_k^2$, then z_i is not a nearest neighbor of q. Next i.
     - Next j.
   - Update the list of nearest neighbors, d^2, and d_k:
     $d_k := \sqrt{d_k^2}$.
   - Next i.

   5.3 Exit function.

Since PAT and DOPAT construct the same search tree, they both require the same amount of preprocessing. However, DOPAT requires more memory because it stores the n_z projections, g_i, in each non-terminal node while PAT only stores the $n_c - 1$ region

boundaries. PAT also requires less memory because it constructs shallower trees with $n_c \geq 2$ subsets at each non-terminal node as opposed to the binary tree constructed by DOPAT. Asymptotically the memory required for PAT scales as $O(n_p)$ and as $O(n_p \log n_p)$ for DOPAT. The difference is usually insignificant for small data sets, but on large data sets PAT will usually require much less memory than DOPAT[2].

3.5 Performance

This section contains a brief summary of the other algorithms included in this study, a description of how the user-specified parameters were chosen, and a description of how the mean query time was estimated.

3.5.1 Algorithms

Algorithm	Source
Bak	Bakamidis [12]
BJS	Baek *et al.* [72]
Cube	Soleymani and Morgera [65]
DOSVQ	Katsavounidis *et al.* [83]
DWCS	Tai and Lin [104, 105]
FN	Fukunaga and Narendra [81]
FNM	Fukunaga and Narendra Modified
K-D Tree	Friedman *et al.* [79]
MiniMax	Cheng *et al.* [13]
OPS	Kim and Park [80]
Poggi	Poggi [66]
TLAESA	Micó *et al.* [82]
WL-AESA	Wu and Lin [103]

Table 3.1: List of algorithms implemented for this study.

Section 3.4 described four of the algorithms implemented for this study. Table 3.1 lists the other thirteen algorithms and the notation used to refer to each algorithm.

Two of the algorithms, K-D Tree and OPS, were modified to incorporate the partial distance search in the terminal nodes, instead of the originally proposed full search. This significantly improves performance. A modification to OPS proposed by Lubiarz and Lockwood [88] was also implemented to guarantee that the exact nearest neighbors are found.

[2]The memory required for DOSVQ, an algorithm proposed by Katsavounidis *et al.* [83], also scales as $O(n_p \log n_p)$.

FNM is a new algorithm that consists of four modifications to FN. First, squared distances were used to avoid the calculation of square roots. Second, the partial distance search was incorporated in the terminal nodes. Third, the elimination criterion used in the terminal nodes was strengthened slightly as described by Equation 3.1. Fourth, a two-stage clustering algorithm was used. The first stage picks initial estimates for the cluster centers by an efficient principal axis splitting method [112]. The second stage is Forgy's K-means clustering algorithm with the stopping criteria proposed by Linde *et al.* [113–115].

3.5.2 User-Specified Parameters

Some of the nearest neighbor algorithms included in this study have a single user-specified parameter. For example, PAT, FN, and FNM all require the user to specify the number of partitions of each non-terminal node, K-D Trees require the user to specify the number of points contained in each terminal node, and Bak and TLAESA require the user to specify the number of anchor points for the triangle-inequality. In most situations, the optimal parameter value depends on the data set distribution and size and it is impractical to perform an exhaustive search to find the best value.

The user-specified parameters were fixed at a constant value for all the results reported here. The parameter values were chosen by comparing the query times on simulated data sets drawn from uniform and normal distributions. The average query times were measured for a range of dimensions, one through twelve, and parameter values ranging from two to fifty. Each data set contained $2,000$ points.

Algorithm	Value
PAT	7
FN	16
FNM	50
TLAESA	3
K-D Tree	40
Bak	2

Table 3.2: Values chosen for algorithms that have a user-specified parameter.

In most cases, the best parameter value varied only slightly with dimension and distribution. For example, Figure 3.3 shows the average query time for PAT for various dimensions and parameter values. Values in the range of five to eight had the lowest query times. In this case, the parameter value was fixed at seven. Table 3.2 lists the parameter values chosen for each of the algorithms that had a user-specified parameter.

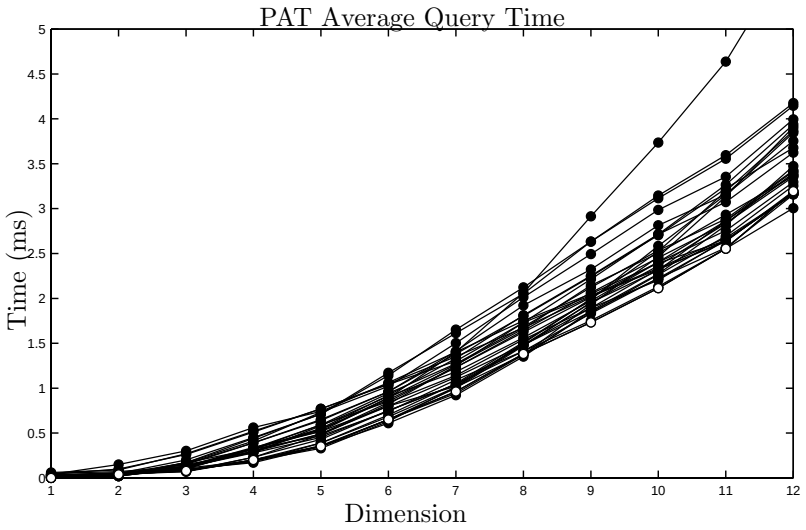


Figure 3.3: Average query time of PAT to find five nearest neighbors as a function of dimension for various numbers of children per non-terminal node. The data set consisted of $2,000$ points drawn from a normal distribution. The white circles show the average query times for seven children per non-terminal node.

3.5.3 Measures of Performance

The purpose of this study is to compare the average query times of the leading algorithms, which is the most critical consideration for many applications. Unfortunately, there is no generally accepted measure of query time; in previous studies researchers have used a wide variety of different measures.

Frequently, researchers report the average number floating-point multiplies, additions, comparisons, and square roots (MACS) per query. This allows the user to determine roughly how fast an algorithm will run on a given architecture, but it does not account for the time required for other operations such as function calls and fixed-point operations. This measure also makes it difficult to compare algorithms, since the performance is architecture-dependent.

Some researchers have estimated or derived the asymptotic performance of their algorithms to determine how the query time scales with dimension (n_d) and number of points (n_p). However, most of these analyses only apply when the n_d-dimensional search space is densely populated, which requires that the number of points scale exponentially with dimension. In practical situations, the data sets are rarely large enough for the asymptotic conditions to apply. These analyses also do not account for the data set distributions, which can strongly affect performance, as discussed in Section 3.3.

For this study, the average query times were measured directly. This approach has the disadvantage that it depends on the characteristics of the hardware and software[3]; but, for lack of a better measure, it is assumed that similar results would be obtained on other architectures.

3.6 Empirical Performance

This section reports the empirical performance of the algorithms on several types of data sets selected from the literature. To verify the algorithms' accuracy, the neighbors found by each algorithm were compared to those found by Brute. Except in cases where different data set points were equidistant to the query point, all of the algorithms found the same neighbors as Brute.

3.6.1 Uniform Distribution

Algorithm	Dimension					Average
	4	8	12	16	20	Rank
PAT	0.163	1.925	6.284	11.462	16.749	3.0
DOSVQ	0.240	2.188	6.630	10.515	14.183	3.0
DOPAT	0.288	2.621	6.519	10.355	14.145	3.4
K-D Tree	0.260	1.825	6.990	11.299	14.824	3.7
PDS	2.649	4.271	6.599	9.571	12.869	5.3
FNM	0.468	2.478	7.165	11.907	15.452	5.6
DWCS	0.841	4.522	9.463	13.930	17.838	7.6
OPS	0.226	3.017	15.114	19.408	22.370	7.7
FN	0.588	3.458	11.514	22.202	29.993	9.1
Poggi	3.220	5.911	8.943	12.203	15.735	9.1
Brute	4.930	9.200	13.252	17.555	21.804	11.1
Cube	2.972	5.946	13.377	24.906	32.524	11.7
BJS	4.003	10.368	16.323	21.396	25.968	12.4
TLAESA	4.807	18.730	27.640	33.639	38.623	13.9
Bak	14.070	20.330	26.078	28.769	31.846	14.7
MiniMax	8.725	15.838	23.865	34.064	43.124	15.0
WL-AESA	86.875	178.055	647.030	1381.135	1767.840	16.7

Table 3.3: Algorithms' average query times (milliseconds) to find the five nearest neighbors at selected dimensions. The data sets contained 5,000 points drawn from a uniform distribution.

The average query times were calculated for data sets and query vectors drawn from a uniform distribution and dimensions ranging from one to twenty. Each data set contained $5,000$ points and the average query times were calculated using $4,000$ queries. For each

[3]The results reported here were generated using a Pentium 200 MHz MMX processor with 512 KB Cache, 128 MB RAM, Windows NT 4.0 SP3, and Visual C++ 6.0 SP1.

dimension, the algorithms were ranked in order of increasing average query times[4]. Table 3.3 shows the average query times at selected dimensions and the average ranks calculated using all twenty dimensions.

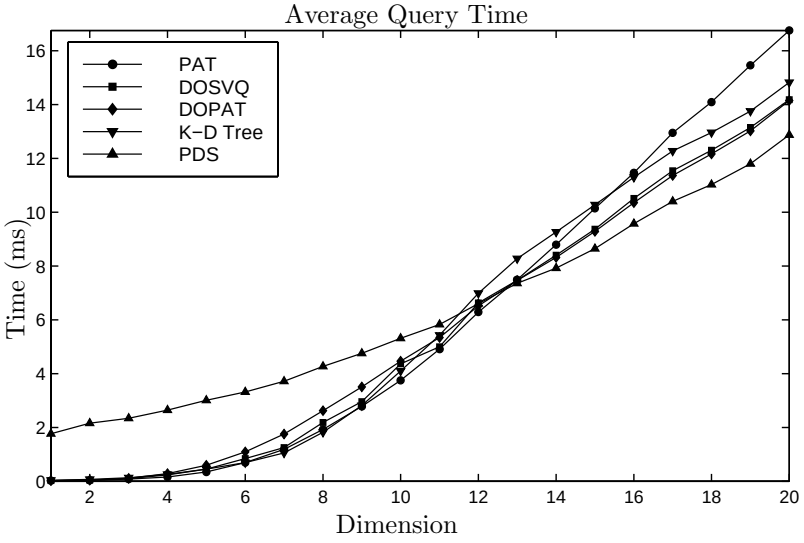


Figure 3.4: Average query time to find five nearest neighbors as a function of dimension. The data sets and query points were drawn from a uniform distribution. The data set contained 5,000 points. Only the best five algorithms are shown.

Figure 3.4 shows a plot of the average query time versus dimension for the top five algorithms. This figure shows the exponential growth in low dimensions and the linear growth in high dimensions that was discussed in Section 3.3.

For this type of data set, PDS performed best in high dimensions because the elimination criteria of the other algorithms are ineffective in high dimensions and have more overhead than PDS. However, in low dimensions, the algorithms that used search trees performed much better than PDS.

According to the average ranking shown in Table 3.3, PAT and DOSVQ were tied for best performance. However, as shown in Figure 3.4, DOSVQ performed better in high dimensions due to a smaller overhead and PAT performed better in low dimensions. DOPAT and K-D Tree also performed very well overall.

The average query times were also measured for various data set sizes. Each data set contained eight variables and the average query times were calculated using 5,000 queries. For each data set size, the algorithms were ranked in order of increasing average query times as shown in Table 3.4.

[4]The ranking was ordered from one to seventeen with a rank of one being the best.

Algorithm	Points						Average
	2,000	5,000	10,000	20,000	50,000	100,000	Rank
K-D Tree	1.20	1.81	2.22	2.80	3.55	4.46	1.3
PAT	1.10	1.93	2.55	3.39	4.08	5.15	2.0
FNM	1.52	2.63	3.59	4.75	7.25	10.62	4.0
DOSVQ	1.04	2.08	3.64	5.51	11.09	20.78	4.2
OPS	2.10	2.99	3.90	4.85	6.08	7.25	4.8
DOPAT	1.21	2.60	4.57	7.97	16.28	28.85	5.8
FN	2.25	3.81	5.47	7.16	10.86	14.64	6.5
DWCS	2.08	4.39	8.20	16.06	36.07	65.73	8.0
PDS	1.96	4.39	8.25	15.81	37.36	71.56	8.3
Cube	2.85	6.01	10.82	19.91	45.12	84.37	10.3
Poggi	2.40	5.94	11.86	23.53	58.36	119.35	10.7
Brute	3.69	9.24	18.75	38.28	96.93	194.05	12.0
BJS	4.11	10.40	21.30	43.44	105.07	206.21	13.0
MiniMax	6.81	15.95	31.79	64.26	161.43	322.38	14.3
TLAESA	8.09	17.80	36.85	69.93	155.56	304.61	14.7
Bak	8.33	22.87	45.75	96.78	258.77	536.21	16.0
WL-AESA	51.03	188.71	495.47	1262.05	4112.76	10621.47	17.0

Table 3.4: Algorithms' average query times (milliseconds) to find the five nearest neighbors as a function of dimension. The data sets were drawn from a uniform distribution in eight dimensions.

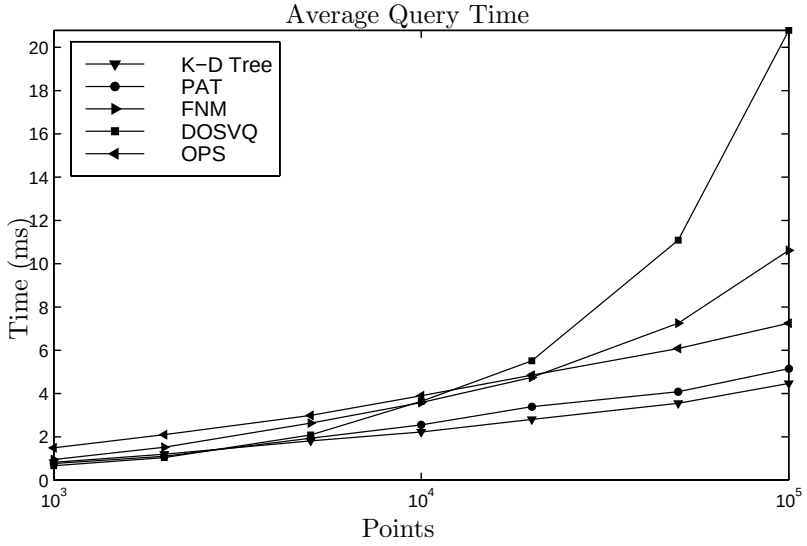


Figure 3.5: Average query time to find five nearest neighbors as a function of the data set size. The data sets and query points were uniformly distributed in eight dimensions. Since the horizontal axis is on a log scale, a straight line indicates that the average query time scales as $O(\log n_p)$ and an exponentially increasing line scales as $O(n_p)$, where n_p is the number of points in the data set. Only the best five algorithms are shown.

Figure 3.5 shows the average query time versus the data set size for the best five algorithms. In this case the average query time of K-D Tree, PAT, and OPS scaled as $O(\log n_p)$ whereas DOSVQ, and FNM scaled as $O(n_p)$. This is an important advantage of PAT, K-D Tree, and OPS. Although the slower algorithms may also eventually scale as $O(\log n_p)$ for larger data sets or lower dimensions, it is significant that PAT, K-D Tree, and OPS achieve logarithmic query time more readily.

3.6.2 Normal Distribution

Algorithm	Dimension					Average
	4	8	12	16	20	Rank
PAT	0.190	2.366	6.800	11.967	16.939	3.0
DOSVQ	0.361	2.999	6.830	10.503	14.258	3.1
DOPAT	0.381	3.092	6.825	10.543	14.238	3.3
K-D Tree	0.326	2.583	7.283	11.457	15.067	3.8
PDS	2.769	4.587	6.990	9.787	13.019	5.2
FNM	1.034	3.418	7.639	11.905	15.553	5.9
DWCS	0.954	4.762	9.539	13.850	17.683	7.1
OPS	0.283	4.366	15.495	19.413	22.465	7.5
Poggi	3.315	6.159	9.158	12.335	15.818	8.8
FN	1.417	5.245	13.502	23.011	30.281	10.3
Brute	4.990	9.326	13.470	17.573	21.892	10.9
Cube	3.117	6.572	13.482	22.360	30.932	11.2
BJS	3.713	10.050	15.993	20.980	25.890	12.1
TLAESA	4.992	18.474	26.958	33.180	38.383	14.2
Bak	15.125	20.830	25.880	29.044	31.638	14.9
MiniMax	8.780	16.078	24.210	33.225	42.596	14.9
WL-AESA	59.785	272.340	683.280	1206.885	1541.315	17.0

Table 3.5: Algorithms' average query times (milliseconds) to find the five nearest neighbors at selected dimensions. The data sets contained 5,000 points drawn from a normal distribution.

The analysis performed using a uniform distribution was repeated using a normal distribution. Table 3.5 shows the average query times at selected dimensions and the average ranks taken over all twenty dimensions. Figure 3.6 shows a plot of the average query time versus dimension for the top five algorithms.

The results are very similar to those obtained using a uniform distribution, which suggests that the relative performance of the algorithms is insensitive to the marginal distributions of data set variables when they are independently distributed.

As with the uniform distribution, PAT had the best overall performance in low and moderate dimensional spaces; other search trees also performed well. PDS had the best performance in high dimensions, though other methods with small overhead had comparable performance.

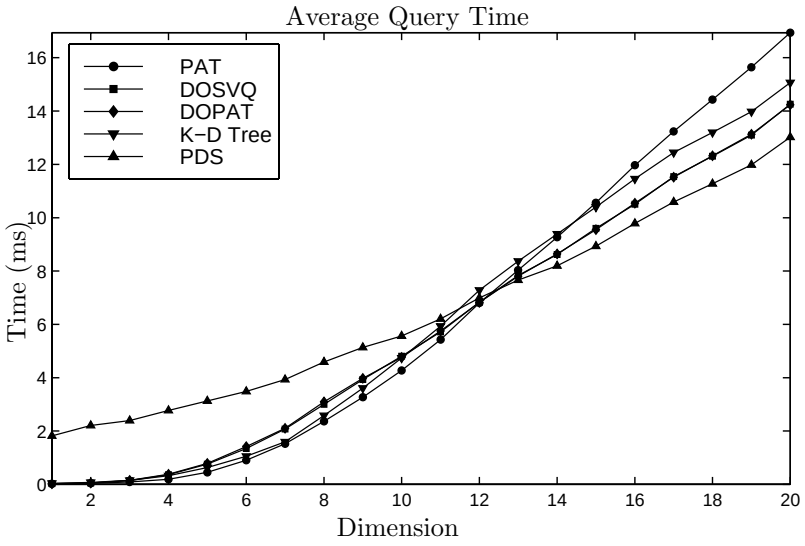


Figure 3.6: Average query time to find five nearest neighbors as a function of dimension. The data sets and query points were drawn from a normal distribution. The data set contained 5,000 points. Only the best five algorithms are shown.

Algorithm	Points						Average
	2,000	5,000	10,000	20,000	50,000	100,000	Rank
PAT	1.26	2.37	3.18	4.61	5.83	7.54	1.0
K-D Tree	1.45	2.66	3.45	4.95	6.38	7.84	2.3
DOSVQ	1.34	3.07	5.36	9.88	21.49	37.30	4.2
DOPAT	1.36	3.14	5.40	10.21	21.31	36.94	4.5
FNM	1.65	3.73	5.67	9.03	16.04	26.01	4.5
OPS	2.69	4.41	5.89	7.71	10.41	12.75	5.0
PDS	2.04	4.60	8.69	16.71	39.58	75.56	7.8
DWCS	2.18	4.88	8.68	17.12	38.05	72.91	7.8
FN	2.71	5.94	9.44	14.58	25.00	41.04	8.2
Poggi	2.38	6.18	11.87	24.66	61.33	120.48	10.2
Cube	3.01	6.51	12.00	21.88	49.55	92.09	10.5
Brute	3.66	9.15	18.72	38.14	96.90	193.95	12.2
BJS	3.84	9.93	19.53	40.54	100.88	187.99	12.8
MiniMax	6.78	16.13	31.89	64.35	161.61	322.86	14.3
TLAESA	7.69	18.28	34.25	67.78	159.35	311.93	14.8
Bak	7.26	21.03	44.82	93.25	278.96	600.74	15.8
WL-AESA	61.05	240.78	686.95	1826.83	5948.47	14553.29	17.0

Table 3.6: Algorithms' average query times (milliseconds) to find the five nearest neighbors as a function of dimension. The data sets were drawn from a normal distribution in eight dimensions.

Table 3.6 shows the average query times for various data set sizes in eight dimensions. The averages were calculated using 5,000 queries. As with the uniform distribution, PAT and K-D Tree achieved logarithmic query time, $O(\log n_p)$. The average query times of DOPAT, DOSVQ, and FNM scaled roughly linearly with the data set size, $O(n_p)$, as shown in Figure 3.7.

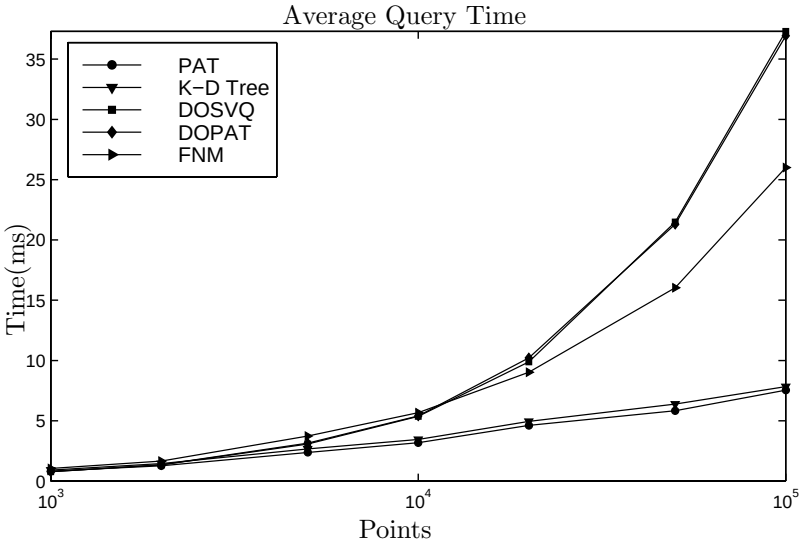


Figure 3.7: Average query time to find five nearest neighbors as a function of the data set size. The data sets and query points were drawn from a normal distribution in eight dimensions. Only the best five algorithms are shown.

3.6.3 Chaotic Time Series

Local models are commonly used for time series prediction [28]. These methods construct simple models using only the neighboring points of the input vector and generally incorporate a fast nearest neighbor algorithm. The models are often constructed to predict one step ahead using a window of previous values as the input. For example, if the time series is represented by $[y_1, y_2, \ldots, y_{n_y}]$, the input vectors are defined as

$$x_t \triangleq [y_t, y_{t-1}, y_{t-2}, \ldots, y_{t-(n_d-1)}]. \tag{3.4}$$

Usually these models are applied to time series generated by nonlinear dynamic systems. If the systems have no external inputs and they meet other very general criteria, the time series are chaotic and all of the points in the data set lie on a low dimensional

manifold[5]. Consequently, the intrinsic dimension of these data sets does not change as the spatial dimension, n_d, is increased.

Algorithm	Dimension						Average
	4	10	16	22	30	40	Rank
DOSVQ	0.054	0.074	0.108	0.144	0.194	0.272	1.4
PAT	0.046	0.074	0.118	0.160	0.244	0.373	2.0
DOPAT	0.050	0.078	0.114	0.194	0.302	0.603	2.7
FNM	0.264	0.344	0.429	0.517	0.663	0.777	4.8
K-D Tree	0.100	0.202	0.354	0.523	0.859	1.520	5.0
FN	0.264	0.360	0.491	0.621	0.653	0.925	5.7
PDS	2.277	2.626	3.263	3.918	5.005	6.884	7.8
TLAESA	0.407	0.543	1.480	5.582	5.410	8.468	7.8
Cube	2.431	3.078	4.270	5.945	9.135	15.380	9.5
DWCS	2.726	7.697	8.102	8.759	9.948	11.681	10.1
OPS	0.144	3.179	13.095	13.614	14.306	14.893	10.4
Poggi	4.469	6.776	8.739	11.867	15.480	20.525	11.6
BJS	4.038	6.902	11.198	15.194	21.317	27.321	12.6
Brute	4.983	11.190	17.772	24.033	32.623	43.302	14.4
WL-AESA	8.759	13.520	18.186	22.633	27.858	36.601	14.6
Bak	9.498	20.620	27.902	35.225	42.697	53.651	16.1
MiniMax	8.861	18.561	28.924	39.060	52.750	70.015	16.6

Table 3.7: Algorithms' average query times (milliseconds) to find the five nearest neighbors at selected dimensions. The data sets contained 5,000 points created from the Lorenz time series. The query points were created from a later section of the same time series.

The Lorenz time series is a common benchmark for measuring the accuracy of time series prediction algorithms[6]. Using this time series, the average query times were calculated for various dimensions using a data set containing $5,000$ points and averaging over $5,000$ queries. Table 3.7 shows the average query times at selected dimensions and Figure 3.8 shows the average query times of the top five algorithms.

In moderate to high dimensional spaces, DOSVQ had the best performance, though PAT and DOPAT also performed very well. Elimination criteria that are insensitive to the spatial dimension of the data set performed much better on this problem than on the independently distributed data sets. For example, TLAESA and FN, both of which rely on the triangle inequality, improved significantly in rank. Although K-D Tree was among the top five algorithms, its query time grew much more quickly with dimension than the other top algorithms because it less able to adapt to the data set distribution, especially when the intrinsic dimension is much smaller than the spatial dimension (see Section 3.3).

[5]Nonlinear manifolds in chaotic time series is discussed in fuller detail in Chapter 6.

[6]The Lorenz time series is described in Chapter 7.

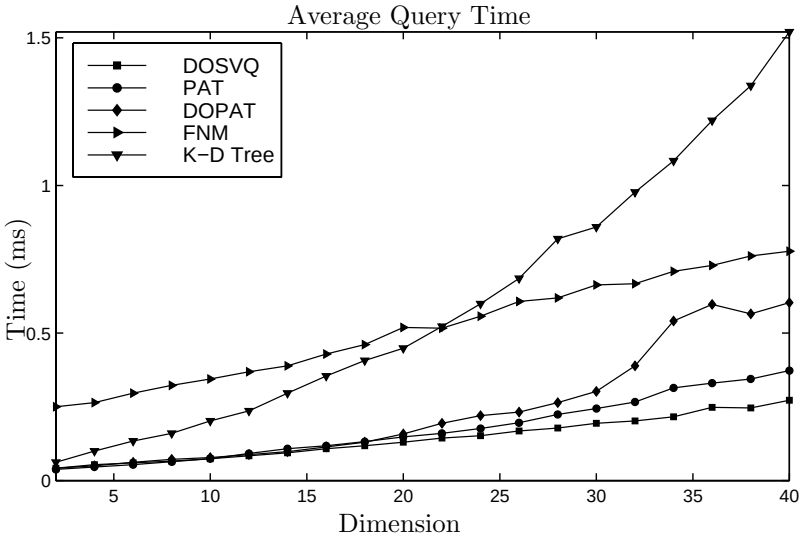


Figure 3.8: Average query time to find five nearest neighbors as a function of dimension. The data sets and query points were created from the Lorenz time series. The data set contained 5,000 points. Only the best five algorithms are shown.

Algorithm	Dimension						Average
	4	10	16	22	30	40	Rank
PAT	0.054	0.124	0.262	0.409	0.597	0.877	1.6
DOSVQ	0.058	0.126	0.238	0.388	0.603	0.961	1.6
DOPAT	0.062	0.146	0.256	0.405	0.661	1.096	2.9
FNM	0.309	0.479	0.611	0.735	0.881	1.136	4.7
K-D Tree	0.112	0.264	0.519	0.945	1.727	2.898	5.0
FN	0.475	0.907	1.218	1.408	1.588	2.019	5.9
TLAESA	0.250	1.316	2.125	3.643	5.041	18.517	7.6
PDS	2.402	2.856	3.441	4.092	5.139	6.668	8.1
DWCS	0.701	3.579	5.718	7.597	9.219	10.427	9.7
Cube	2.604	3.339	4.474	6.037	8.795	13.259	9.8
Poggi	3.391	5.055	5.626	6.798	8.879	11.797	10.8
OPS	0.182	3.485	12.859	13.495	14.475	14.998	11.1
BJS	3.253	5.916	7.729	9.860	14.291	20.129	12.4
Brute	4.981	11.258	17.425	23.874	32.573	43.188	13.9
Bak	14.020	20.229	24.423	31.003	39.545	50.208	15.7
MiniMax	8.722	18.605	28.571	38.994	53.110	70.355	16.1
WL-AESA	10.936	14.741	23.358	33.054	67.376	93.252	16.2

Table 3.8: Algorithms' average query times (milliseconds) to find the five nearest neighbors at selected dimensions. The data sets contained 5,000 points created from the Santa Fe time series. The query points were created from a later section of the same time series.

This analysis was repeated using the Santa Fe laser time series, another common time series prediction benchmark[7]. The average query times were measured for various dimensions using a data set containing $5,000$ points and averaging over $5,000$ queries. Table 3.8 shows the average query times at selected dimensions and Figure 3.9 shows the average query times of the top five algorithms.

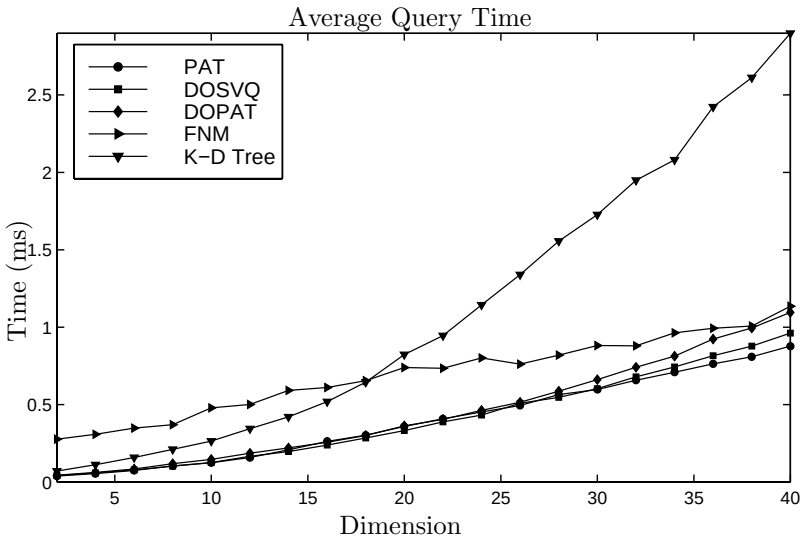


Figure 3.9: Average query time to find five nearest neighbors as a function of dimension. The data sets and query points were created from the Santa Fe time series. The data set contained $5,000$ points. Only the best five algorithms are shown.

In this case, PAT and DOSVQ were tied for best performance. DOPAT and FNM also performed very well.

3.6.4 Vector Quantization

Vector quantization is a popular and powerful method of compression. It begins with the construction of a codebook, which is typically constructed by applying a clustering algorithm to a large, representative data set of the signals to be compressed. Once the codebook is created, the signal to be compressed is divided into blocks, or vectors, of a user-specified length. The nearest neighbor in a codebook is found for each vector and the index of the nearest neighbor is used to encode, or represent, the vector. In a communications application, the sender and the receiver are both given a copy of the codebook. Only the indices of the nearest neighbors are transmitted. Once the receiver receives an index, the corresponding vector in the codebook is retrieved and used to represent the original vector.

[7]The Santa Fe time series is discussed in Chapter 7.

This technique is commonly used to compress speech and images. One of the practical limitations of this method is the large computational cost of finding the nearest neighbor in the codebook.

To test the performance of the nearest neighbor algorithms in vector quantization encoding applications, a range of different codebooks were constructed for a speech signal and three gray-scale images that are commonly used as benchmarks for vector quantization algorithms[8]. The codebooks were constructed using the clustering algorithm described by Franti *et al.* [112].

Table 3.9 shows the average query times, calculated using 35,000 queries, to find the nearest neighbor for the speech signal. Tables 3.10, 3.11, and 3.12 show the average query times for the benchmark images *Lena*, *Boat*, and *Baboon*[9], respectively. For the images, the average query times were calculated using 20,000 queries for the 2×2 image blocks and 15,000 queries for the 4×4 image blocks. In all cases, the query vectors were taken from the original signals used to construct the codebook.

Algorithm	8 Dimensions				16 Dimensions				Rank
	1,024	2,048	4,096	8,192	1,024	2,048	4,096	8,192	
PAT	0.058	0.075	0.097	0.119	0.207	0.278	0.380	0.441	1.4
DOSVQ	0.060	0.080	0.108	0.159	0.181	0.251	0.371	0.488	1.6
DOPAT	0.071	0.100	0.142	0.220	0.236	0.352	0.573	0.890	3.0
K-D Tree	0.196	0.231	0.280	0.355	0.634	0.875	1.211	1.500	4.1
FNM	0.335	0.520	0.848	1.307	0.571	0.944	1.621	2.637	4.9
FN	0.575	0.804	1.359	2.012	1.375	1.894	3.148	4.896	6.9
PDS	0.585	1.127	2.076	4.354	0.876	1.603	3.084	5.891	7.1
BJS	0.612	1.187	2.248	4.386	1.099	2.032	3.927	7.458	8.4
Cube	0.725	1.332	2.421	4.922	1.492	2.567	4.619	8.336	9.9
Poggi	0.829	1.570	3.121	7.187	1.147	2.158	4.487	9.359	10.1
OPS	0.931	1.064	1.354	1.901	2.522	5.074	10.647	22.442	10.8
TLAESA	1.199	2.567	2.751	3.946	1.958	4.161	7.699	11.584	11.8
DWCS	1.232	2.363	4.607	8.812	1.765	3.440	6.867	14.445	12.3
WL-AESA	1.080	2.283	4.651	9.532	2.061	3.973	8.199	15.894	12.9
Brute	1.885	3.772	7.337	15.408	3.528	7.018	14.418	29.327	15.3
Bak	1.938	4.132	8.482	18.849	3.383	6.880	14.826	29.724	15.8
MiniMax	3.091	6.209	12.360	25.546	5.817	11.568	23.667	47.801	17.0

Table 3.9: Algorithms' average query times (milliseconds) to find the nearest neighbor for a codebook constructed from a speech signal. The results are shown for four codebook sizes and two vector sizes.

[8] At the time of writing, the speech signal was available at *http://wavcentral.simplenet.com/sounds/-televis/cbrown/wetsuck.wav* and the images were available at *http://meru.cecs.missouri.edu/SLCCA/-slcca.html*.

[9] This image is sometimes called *Mandrill*.

Algorithm	2 × 2 Pixels				4 × 4 Pixels				Rank
	1,024	2,048	4,096	8,192	1,024	2,048	4,096	8,192	
DOSVQ	0.026	0.034	0.042	0.054	0.073	0.102	0.136	0.160	1.5
PAT	0.025	0.032	0.040	0.047	0.087	0.128	0.167	0.194	1.6
DOPAT	0.028	0.037	0.047	0.064	0.083	0.132	0.187	0.253	2.9
K-D Tree	0.052	0.067	0.080	0.107	0.232	0.340	0.465	0.564	4.0
FNM	0.130	0.251	0.386	0.548	0.268	0.547	0.839	1.341	5.5
FN	0.209	0.352	0.540	0.969	0.608	0.946	1.445	2.069	6.8
TLAESA	0.245	0.447	0.756	1.318	0.538	1.080	1.929	2.500	7.4
PDS	0.428	0.880	1.793	3.658	0.601	1.142	2.291	4.569	8.9
OPS	0.079	0.102	0.126	0.141	2.214	4.702	9.979	22.124	9.3
DWCS	0.363	0.607	0.977	1.602	1.311	2.911	5.482	10.320	10.1
Cube	0.477	0.971	1.955	3.918	0.874	1.596	3.048	5.697	10.3
WL-AESA	0.473	1.024	2.159	4.505	0.933	1.836	3.285	5.517	10.8
BJS	0.671	1.320	2.523	5.818	1.319	2.606	5.491	12.542	12.4
Poggi	0.868	1.732	3.394	7.273	2.326	4.732	10.048	21.120	14.0
Bak	0.997	2.337	4.993	10.580	3.178	6.776	12.194	19.773	15.3
Brute	0.973	1.990	4.073	8.201	3.499	6.949	14.325	29.312	15.5
MiniMax	1.660	3.409	6.933	13.945	5.733	11.360	23.320	47.280	17.0

Table 3.10: Algorithms' average query times (milliseconds) to find the nearest neighbor for a codebook constructed from the image *Lena*. The results are shown for four codebook sizes and two vector sizes.

Algorithm	2 × 2 Pixels				4 × 4 Pixels				Rank
	1,024	2,048	4,096	8,192	1,024	2,048	4,096	8,192	
PAT	0.024	0.030	0.038	0.043	0.092	0.136	0.175	0.182	1.5
DOSVQ	0.025	0.031	0.042	0.050	0.079	0.109	0.144	0.143	1.5
DOPAT	0.026	0.034	0.046	0.061	0.093	0.138	0.198	0.238	3.0
K-D Tree	0.052	0.063	0.077	0.106	0.268	0.391	0.529	0.504	4.0
FNM	0.138	0.244	0.383	0.591	0.301	0.545	0.920	1.276	5.5
FN	0.206	0.320	0.547	0.775	0.594	1.045	1.622	2.242	6.5
TLAESA	0.257	0.545	1.061	1.339	0.654	1.194	2.638	3.552	7.9
PDS	0.478	0.876	1.785	3.625	0.631	1.264	2.466	4.735	8.8
OPS	0.075	0.098	0.131	0.155	2.262	4.878	10.679	21.313	9.5
Cube	0.524	0.965	1.945	3.883	0.969	1.830	3.375	6.107	10.0
DWCS	0.362	0.548	0.930	1.398	1.377	3.098	5.985	8.379	10.3
WL-AESA	0.555	1.223	2.530	5.375	1.090	2.083	4.040	6.791	11.1
BJS	0.695	1.286	2.544	5.025	1.356	2.606	5.021	9.995	12.0
Poggi	0.951	1.770	3.530	7.826	2.341	4.780	9.721	18.666	13.8
Brute	1.035	1.983	4.033	8.173	3.392	7.035	14.312	29.211	15.4
Bak	0.962	2.168	4.783	10.371	3.478	7.252	13.633	17.061	15.4
MiniMax	1.716	3.393	6.945	13.976	5.568	11.486	23.226	47.267	17.0

Table 3.11: Algorithms' average query times (milliseconds) to find the nearest neighbor for a codebook constructed from the image *Boat*. The results are shown for four codebook sizes and two vector sizes.

Algorithm	2 × 2 Pixels				4 × 4 Pixels				Rank
	1,024	2,048	4,096	8,192	1,024	2,048	4,096	8,192	
PAT	0.037	0.043	0.050	0.053	0.306	0.445	0.517	0.380	1.5
DOSVQ	0.034	0.048	0.058	0.069	0.273	0.435	0.557	0.445	1.6
DOPAT	0.037	0.050	0.062	0.077	0.282	0.449	0.582	0.484	2.9
K-D Tree	0.074	0.091	0.103	0.120	0.712	1.160	1.503	1.291	4.3
FNM	0.175	0.271	0.449	0.655	0.565	1.078	1.631	2.227	5.3
FN	0.248	0.402	0.654	0.810	1.152	1.955	3.199	4.583	7.1
PDS	0.529	0.991	1.896	3.693	1.020	1.901	3.487	6.244	8.3
DWCS	0.243	0.392	0.623	0.967	1.756	3.719	6.665	9.430	8.3
TLAESA	0.382	0.627	1.041	2.524	1.784	3.071	5.766	10.086	8.9
OPS	0.112	0.133	0.155	0.170	2.585	5.587	10.942	22.015	9.3
Cube	0.576	1.080	2.056	3.988	1.925	3.397	5.804	9.784	10.3
BJS	0.625	1.218	2.283	4.444	1.895	3.647	6.715	12.400	11.3
Poggi	0.761	1.527	2.934	6.244	2.083	4.165	8.197	15.953	12.6
WL-AESA	1.143	2.526	5.347	11.330	3.110	5.980	9.057	13.128	14.1
Brute	1.054	2.062	4.084	8.108	3.544	7.136	14.417	29.171	14.6
Bak	1.477	3.110	7.355	14.933	4.086	9.154	17.630	27.367	16.1
MiniMax	1.756	3.561	6.925	13.972	5.875	11.752	23.458	47.458	16.8

Table 3.12: Algorithms' average query times (milliseconds) to find the nearest neighbor for a codebook constructed from the image *Baboon*. The results are shown for four codebook sizes and two vector sizes.

The results are similar to those obtained for chaotic time series: PAT and DOSVQ were statistically tied for the best performance. DOPAT, K-D Tree, and FNM also performed well.

3.7 Discussion

This chapter described two new fast nearest neighbor algorithms: the principal axis tree (PAT) and the depth-only principal axis tree (DOPAT). The performance of these algorithms was compared to fifteen other fast nearest neighbor algorithms on a variety of benchmark problems, including independently-distributed simulated data sets, chaotic time series, and vector quantization codebooks.

Some of the algorithms used in this study had a single user-specified parameter. These parameters were fixed at the value that optimized the overall performance on independently-distributed data sets ranging from one to twelve dimensions. Better performance may be achievable on specific problem domains if different parameter values are used.

Better performance would also have been achieved if general purpose improvements had been used. For example, in many applications nearest neighbor algorithms are applied to a set of query points that are highly correlated. Many researchers have independently proposed using the nearest neighbors of the previous query as initial candidates for the

nearest neighbors of the current query [5, 54, 66, 67, 74]. This type of improvement can be applied to any nearest neighbor algorithm, but it was not used here.

The algorithms that performed best incorporated search trees and partial distance search (PDS). For the independently-distributed data sets in high dimensions, PDS performed best due to the small overhead of the elimination criteria.

Other algorithms also performed well. DOPAT frequently achieved the third best performance, after PAT and DOSVQ, and was always among the top-ranked algorithms. K-D Tree, modified to include PDS, also performed very well on all the problems. FNM, performed nearly as well as K-D Tree, and in some cases much better.

Overall, PAT and DOSVQ [83] performed the best. The impressive performance of DOSVQ is surprising since it has received scant attention in the literature.

In many cases, the average query times of PAT and DOSVQ were statistically indistinguishable. In other cases, the difference was significant. For example, DOSVQ had better performance on independently-distributed data sets in high dimensions, mostly due to the smaller overhead of its elimination criteria. PAT had superior performance on large independently-distributed data sets and scaled as $O(\log n_p)$ as opposed to $O(n_p)$ for DOSVQ, where n_p is the number of points in the data set. PAT also requires less memory, $O(n_p)$, than DOSVQ, $O(n_p \log n_p)$. Consequently, PAT should have a significant advantage on larger data sets than were used in this study.

3.8 Summary

The fast nearest neighbor algorithms described in this chapter remove a significant barrier to using local models on large data sets—the large computational cost of finding the nearest neighbors. The next chapter discusses local linear regularization, another important issue for local modeling.

Chapter 4

Local Linear Regularization

I have traveled the length and breadth of this country and talked with the best people, and I can assure you that data processing is a fad that won't last out the year.
—The editor in charge of business books for Prentice Hall, 1957

Accurate local linear models can be difficult to create because they must be constructed with only a small neighborhood of points. Often, the number of points will be nearly equal to the degrees of freedom, which makes it hard to estimate the linear coefficients accurately. This is especially problematic when the points are collinear.

To alleviate this problem researchers have introduced different methods of regularization, which bias the model coefficients to reduce the output variance and thereby increase stability. This chapter describes two of the most popular and effective regularization methods: ridge regression and principal components regression[1]. The introduction to each of these methods is followed by a discussion of how they can each be generalized to work with local weighting, introduced in Chapter 2. Ridge regression is also generalized to use a vector of ridge coefficients, which enables the user to assign a different penalty to each model coefficient. Principal components regression is generalized further to use weighted centering and smooth thresholds.

This chapter is organized as follows. Section 4.1 introduces the motivation for regularization. Section 4.2 describes ridge regression and a generalization of this method called weighted vectored ridge regression. Section 4.3 introduces principal components regression

[1]For a brief review of other regularization methods see Kugiumtzis *et al.* [44].

and several generalizations of this method. Section 4.4 discusses the tradeoffs between the two new methods of regularization.

4.1 Introduction

In Chapter 2 it was shown that the optimal linear model coefficients, in the least squares sense, are given by

$$\nu = (A^{\mathrm{T}} A)^{-1} A^{\mathrm{T}} b, \tag{4.1}$$

where $A \in \mathbb{R}^{k \times n_d+1}$ is a matrix that contains the k neighboring points, each appended with a constant value of one, and $b \in \mathbb{R}^k$ is a vector that contains the corresponding target values,

$$A \triangleq \begin{bmatrix} x_1^{\mathrm{T}} & 1 \\ x_2^{\mathrm{T}} & 1 \\ \vdots & \\ x_k^{\mathrm{T}} & 1 \end{bmatrix}, \qquad b \triangleq \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix}.$$

This is often called the ordinary least squares (OLS) solution.

If columns of the A matrix are nearly collinear, the the matrix product $A^{\mathrm{T}}A$ will be nearly singular and some of the model coefficients will be very large, which, in turn, causes the model outputs to be very large. This may not occur if the model input vector, q, happens to lie in the column space of A, but this rarely happens in practice.

The two methods of regularization described in this chapter replace the matrix product $A^T A$ with a matrix that is better conditioned. Although this results in a biased solution[2] it also reduces the output variance and the expected squared error is often decreased as a result.

[2]The OLS solution given by Equation 4.1 minimizes the *sample* sum of squared errors, where the sum is taken over the k neighboring points. If the sample is small, which is often the case, it may be a poor approximation of the *expected* squared error. Biasing the solution will increase the sample sum of squared errors, but it may decrease the expected squared error.

4.2 Ridge Regression

Ridge regression replaces the matrix product $A^{\mathrm{T}}A$ with $(A^{\mathrm{T}}A + \rho I)$ where ρ is a scalar and $I \in \mathbb{R}^{(n_d+1)\times(n_d+1)}$ is the identity matrix. The least squares model coefficients are given by

$$\nu = \left(A^{\mathrm{T}}A + \rho I\right)^{-1} A^{\mathrm{T}}. \tag{4.2}$$

These coefficients minimize the penalized sum of squared errors, which can be derived by following the same steps as the derivation of the OLS solution given in Chapter 2. The penalized sum of squared errors is given by

$$\begin{aligned}
\mathrm{P}_{\mathrm{RR}}(\nu) &\triangleq \sum_{i=1}^{k} (y_{c_i} - \hat{y}_{c_i})^2 + \rho \sum_{i=1}^{n_d+1} \nu_i^2, \\
&= \sum_{i=1}^{k} \left(y_{c_i} - \tilde{x}_{c_i}^{\mathrm{T}}\nu\right)^2 + \rho \sum_{i=1}^{n_d+1} \nu_i^2, \\
&= (b - A\nu)^{\mathrm{T}}(b - A\nu) + \rho\, \nu^{\mathrm{T}}\nu,
\end{aligned}$$

where $\tilde{x}_i$ denotes the ith nearest vector appended with the constant 1, $\tilde{x}_i^{\mathrm{T}} \triangleq [x_i^{\mathrm{T}}\ \ 1]$.

The parameter ρ controls the weight that is given to minimizing the model coefficients versus the sum of squared errors. In the limit as $\rho \to \infty$, the model coefficients' magnitude all approach zero, which reduces the second moment of the model output to zero, $\mathrm{E}[\hat{y}^2] \to 0$. At the other extreme, as $\rho \to 0$, the model coefficients approach the OLS solution and the bias is reduced to zero: $\mathrm{E}\left[||\nu_{\mathrm{OLS}} - \nu_{\mathrm{RR}}||^2\right] \to 0$, where ν_{OLS} is given by Equation 4.1 and ν_{RR} is given by Equation 4.2.

The value for ρ is usually specified by the user, though Golub *et al.* have suggested a method of picking ρ so as to minimize the cross-validation weighted sum of squared errors [116][3].

A more thorough discussion on ridge regression is available in most books on linear regression, such as Neter *et al.* [117].

4.2.1 Weighted Vectored Ridge Regression (WVRR)

This section describes two generalizations of ridge regression. First, the sum of squared errors are generalized to the weighted case. Second, the penalty term is generalized to a vector

[3]The results of this work are summarized in Golub and Van Loan [111, pp. 583–5], the source of this citation.

so that there is a separate penalty term for each local model input. This technique, termed weighted vectored ridge regression (WVRR), minimizes the following penalty function:

$$\begin{aligned} \mathrm{P}_{\mathrm{WVRR}}(\nu, r) &\triangleq \sum_{i=1}^{k} w_i^2 \left(y_{c_i} - \hat{y}_{c_i}\right)^2 + \sum_{i=1}^{n_d+1} r_i^2 \nu_i^2, \\ &= \sum_{i=1}^{k} w_i^2 \left(y_{c_i} - \tilde{x}_{c_i}^{\mathrm{T}} \nu\right)^2 + \sum_{i=1}^{n_d+1} r_i^2 \nu_i^2, \\ &= (b - A\nu)^{\mathrm{T}} W^2 (b - A\nu) + \nu^{\mathrm{T}} R^2 \nu, \end{aligned} \tag{4.3}$$

where $w \in \mathbb{R}^k$ is a vector of the penalty weights and $r \in \mathbb{R}^{n_d+1}$ is a vector of the ridge coefficients. The matrices W and R are given by,

$$W \triangleq \mathrm{diag}(w), \quad = \begin{bmatrix} w_1 & 0 & \cdots & 0 \\ 0 & w_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & w_k \end{bmatrix}, \qquad R \triangleq \mathrm{diag}(r), \quad = \begin{bmatrix} r_1 & 0 & \cdots & 0 \\ 0 & r_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & r_{n_d+1} \end{bmatrix}.$$

The local linear model coefficients, ν, that minimize the weighted vectored ridge regression penalty $\mathrm{P}_{\mathrm{WVRR}}(\nu, r)$, are derived in Appendix A. The solution is repeated below:

$$\nu = \left(A^T W^2 A + R^2\right)^{-1} A^{\mathrm{T}} W^2 b. \tag{4.4}$$

4.2.2 Discussion

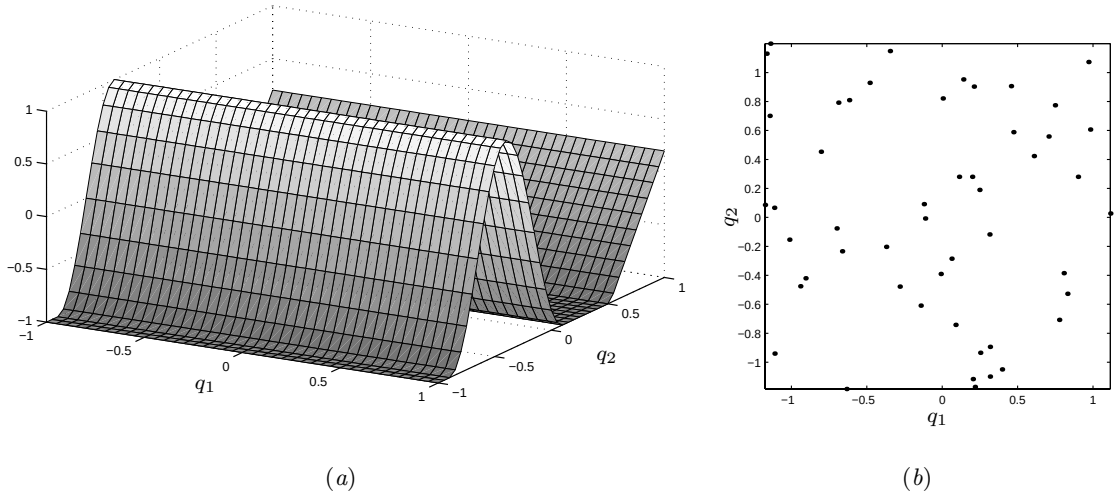
Using a different penalty weight for each model input substantially increases the flexibility of ridge regression. In many cases, one or more of the observed variables are not correlated with the process output and should be ignored. Although the ordinary least squares solution should assign small values to these coefficients, the use of small neighborhoods severely limits the accuracy of the estimated coefficients and they may not be nearly as small as they should be. If the user knows that some of the inputs should have less influence on the model output, vectored ridge regression allows the solution to be biased to reflect this knowledge explicitly.

For all of the results reported here, the last ridge parameter that corresponds to the constant input was set equal to zero, $r_{n_d+1} = 0$. This ensures that in the limit as all of

the other ridge parameters approach infinity, $v_i \to \infty$ for all $i \leq n_d$, the solution becomes equivalent to the weighted average.

Like ordinary ridge regression, vectored ridge regression can also be understood as a means of improving the conditioning of the matrix product $A^{\mathrm{T}}A$. Although it is possible that its replacement, $A^{\mathrm{T}}A + R^2$, is more ill-conditioned than the original matrix product, this does not occur in practice. Generally the addition of R, a positive semidefinite diagonal matrix, improves the conditioning of the matrix inversion.

4.2.3 Example



(a) (b)

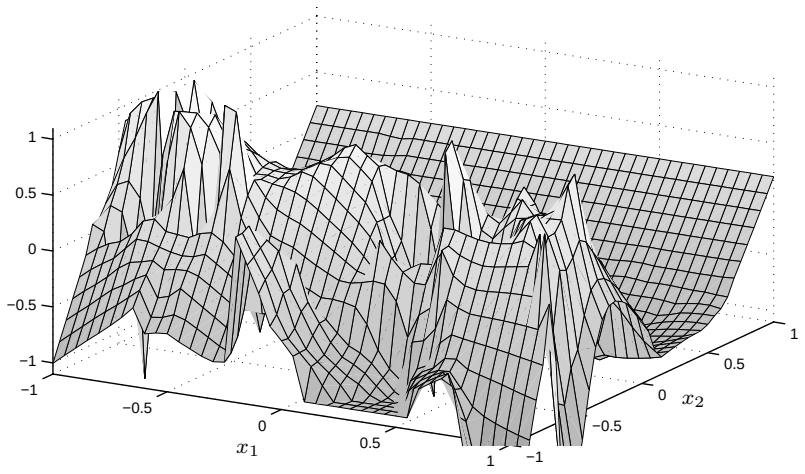
Figure 4.1: (a) Surface plot of the RampHill1D function. (b) Scatter plot of the input variables for the RampHill1D data set.

To illustrate the potential of WVRR a modified version of the RampHill data set introduced in Chapter 2 was used in which the input q_1 was held fixed at $q_1 = -0.40$. The RampHill1D function and a scatter plot of the input variables are shown in Figure 4.1.

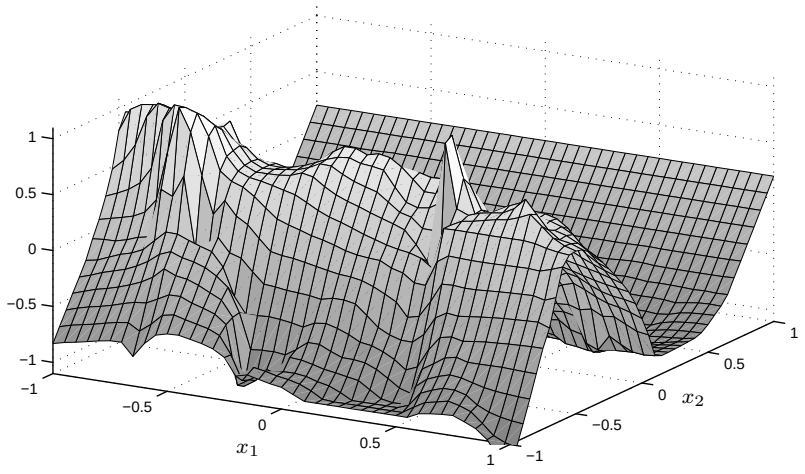
Figure 4.2a illustrates the weighted ordinary least squares solution for a local linear model[4]. In Figure 4.2b, the influence of q_1 on the local model was eliminated by setting the corresponding ridge coefficient equal to four, $r_1 = 4$, while the other ridge coefficient was left as zero, $r_2 = 0$. The resulting model output is much closer to the true function.

The model would have been more accurate if the distance metric had also been chosen to ignore q_1. The weighted Euclidean metric, introduced in Chapter 1, is capable of this

[4]This is equivalent to setting the ridge coefficients to zero.



(a) $r = [0 \quad 0]^{\mathrm{T}}$



(b) $r = [4 \quad 0]^{\mathrm{T}}$

Figure 4.2: The output of a local averaging model for two different ridge vectors and $k = 4$. Figure (a) shows the ordinary least squares solution. The ridge vector in Figure (b) effectively eliminates the influence of x_1 on the local linear model.

and a method for optimizing the metric coefficients and ridge coefficients is discussed in Chapter 5.

4.3 Principal Components Regression

Principal components regression is another method for coping with ill-conditioning in the matrix product, $A^{\mathrm{T}}A$. This technique relies on the singular value decomposition (SVD) of $A_W \triangleq WA$, given by

$$A_W = U\Sigma V^{\mathrm{T}},$$

where $U \in \mathbb{R}^{k\times k}$ and $V \in \mathbb{R}^{(n_d+1)\times(n_d+1)}$ are orthonormal, $U^{\mathrm{T}}U = UU^{\mathrm{T}} = I$ and $V^{\mathrm{T}}V = VV^{\mathrm{T}} = I$, and Σ is a diagonal positive semidefinite matrix, $\Sigma = \mathrm{diag}(\sigma)$ where $\sigma_i \geq \sigma_{i+1} \geq 0$ for all i. The elements of the vector σ are called the singular values of the matrix A_W.

Using this decomposition, the weighted ordinary least squares solution can be written as

$$\begin{aligned}
\nu &= \left(A^{\mathrm{T}}W^2A\right)^{-1}A^{\mathrm{T}}W^2b, \\
&= \left(A_W^{\mathrm{T}}A_W\right)^{-1}A_W^{\mathrm{T}}b_W, \\
&= V\Sigma^{-1}U^{\mathrm{T}}b_W,
\end{aligned}$$

where $b_W \triangleq Wb$. The output of the linear model for a query vector q is given by,

$$\begin{aligned}
\hat{y}(q) &= \tilde{q}^{\mathrm{T}}\nu, \\
&= \tilde{q}^{\mathrm{T}}V\Sigma^{-1}U^{\mathrm{T}}b_W, \\
&= \sum_{i=1}^{\min(k,n_d+1)} (\tilde{q}^{\mathrm{T}}v_i)\left(\frac{1}{\sigma_i}\right)(u_i^{\mathrm{T}}b_W),
\end{aligned}$$

where $\tilde{q}^{\mathrm{T}} \triangleq [q^{\mathrm{T}}\ \ 1]$, v_i denotes the ith column of V, and u_i denotes the ith column of U. The parentheses have been used to emphasize that each element in the sum is a product of three scalars.

4.3.1 Truncated Principal Components Regression

If the columns of the A_W matrix are nearly collinear, one or more of the singular values, σ_i, will be nearly zero. Since each element of the summation is multiplied by $\frac{1}{\sigma_i}$, the

elements with small singular values can cause the model output to be unreasonably large; or, equivalently, the variance of the model output to increase. To counter this effect researchers often truncate the summation,

$$\hat{y}(q) = \sum_{i=1}^{n_\sigma} (q^{\mathrm{T}} v_i) \left(\frac{1}{\sigma_i}\right) (u_i^{\mathrm{T}} b_W), \tag{4.5}$$

where $n_\sigma < \min(k, n_d+1)$. This technique, called truncated principal components regression (TCPR), has the effect of smoothing the model output.

4.3.2 Centering

Researchers often use a slight variation on the method described in the previous section called centering. To introduce the motivation for centering, it is useful to consider the unweighted case more carefully. The generalization of centering to the weighted case is discussed in the next section.

For unweighted ordinary least squares, the model coefficients are given as

$$\nu = (A^{\mathrm{T}} A)^{-1} A^{\mathrm{T}} b.$$

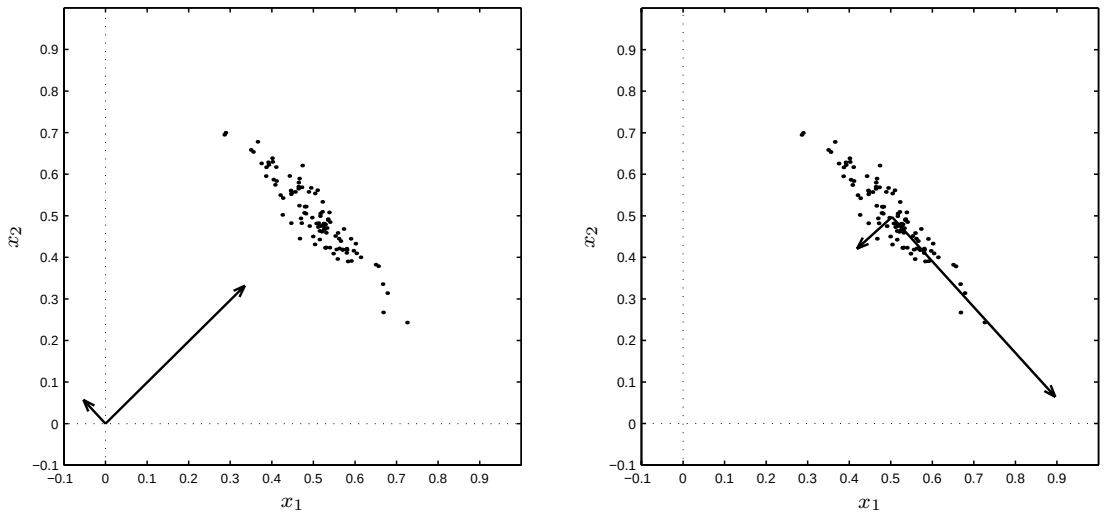
If singular value decomposition is used to decompose the matrix $A = U\Sigma V^{\mathrm{T}}$, the solution can be written as

$$\nu = \left(V\Sigma^2 V^{\mathrm{T}}\right)^{-1} A^{\mathrm{T}} b.$$

This is equivalent to performing an inversion of $A^{\mathrm{T}}A$ by an eigenvalue decomposition, $A^{\mathrm{T}}A = V\Sigma^2 V^{\mathrm{T}}$. Since the matrix product $A^{\mathrm{T}}A$ is real and symmetric, the eigenvectors are real and orthonormal and the eigenvalues are real and nonnegative. Without loss of generality, the eigenvalues can be arranged such that $\sigma_i \geq \sigma_{i+1}$. Using this convention, the eigenvectors represent the directions of maximum amplification by the matrix A

$$v_i = \underset{\substack{q^{\mathrm{T}}q=1,\\ q^{\mathrm{T}}v_j=0\,\forall j<i}}{\operatorname{argmax}} \; q^{\mathrm{T}} A^{\mathrm{T}} A q.$$

The eigenvectors are also called the principal components of the matrix A.



(a) Without Centering (b) With Centering

Figure 4.3: The vectors point in the direction of the principal components. Each vector is scaled so that the vector length is proportional to the corresponding singular value. (a) The principal components without centering. (b) The principal components after centering.

Statistically, the principal components can be understood as the vectors that maximize the sum of the estimated second joint moments of the elements of q and the neighboring variables,

$$\begin{aligned} q^{\mathrm{T}} A^{\mathrm{T}} A q &= \sum_{i=1}^{n_d} \sum_{j=1}^{k} (q_i x_{j,i})^2 + 1, \\ &\approx \sum_{i=1}^{n_d} k \mathrm{E}\left[(q_i x_{\cdot,i})^2\right] + k, \end{aligned}$$

where $\mathrm{E}[\cdot]$ denotes expectation and $x_{\cdot,i}$ is a random variable that represents the ith model input. If the k neighboring points have large sample means, the first few principal components may not represent the directions of maximum *variance*. For example, Figure 4.3a shows the two principal components for a synthetic data set generated from a normal distribution. Since the data set has a large mean, the largest principal component points in a direction that is orthogonal to the direction of maximum variance.

To correct this problem, it is common practice to subtract the sample mean from each of the neighboring points prior to the singular value decomposition,

$$\tilde{A} \triangleq \begin{bmatrix} x_1^{\mathrm{T}} - \bar{x}^{\mathrm{T}} \\ x_2^{\mathrm{T}} - \bar{x}^{\mathrm{T}} \\ \vdots \\ x_k^{\mathrm{T}} - \bar{x}^{\mathrm{T}} \end{bmatrix},$$

where $\bar{x}$ is the sample mean of the k neighboring points,

$$\bar{x} \triangleq \frac{1}{k} \sum_{i=1}^{k} x_i. \tag{4.6}$$

The model output is then given by,

$$\begin{aligned} \hat{y}(q) &= (q - \bar{x})^{\mathrm{T}} \tilde{V} \tilde{\Sigma}^{-1} \tilde{U}^{\mathrm{T}} b + \bar{b}, \\ &= \bar{b} + \sum_{i=1}^{\min(k, n_d)} ((q - \bar{x})^{\mathrm{T}} \tilde{v}_i) \left(\frac{1}{\tilde{\sigma}_i} \right) (\tilde{u}_i^{\mathrm{T}} b), \end{aligned} \tag{4.7}$$

where the singular value decomposition of $\tilde{A}$ is denoted as $\tilde{A} = \tilde{U} \tilde{\Sigma} \tilde{V}^{\mathrm{T}}$ and $\bar{b}$ is the sample mean of b,

$$\bar{b} = \frac{1}{k} \sum_{i=1}^{k} b_i.$$

This technique is called centering. Note that centering eliminates the need to append a one to each of the input vectors.

Intuitively, the principal components of the matrix $\tilde{A}$ maximize the sum of the estimated covariances of the elements of q and the neighboring variables,

$$\begin{aligned} (q - \bar{x})^{\mathrm{T}} \tilde{A}^{\mathrm{T}} \tilde{A} (q - \bar{x}) &= \sum_{i=1}^{n_d} \sum_{j=1}^{k} (q_i - \bar{x}_i)(x_{j,i} - \bar{x}_i), \\ &\approx \sum_{i=1}^{n_d} k \mathrm{E} \left[(q_i - \mathrm{E}[q_i]) \left(x_{\cdot,i} - \mathrm{E}[x_{\cdot,i}] \right) \right], \\ &= \sum_{i=1}^{n_d} k \operatorname{cov}(q_i, x_{\cdot,i}), \end{aligned}$$

where $\operatorname{cov}(\cdot, \cdot)$ denotes covariance.

Figure 4.3*b* shows the principal components of a two-dimensional data set drawn from a normal distribution after centering. In this case, the largest principal component points in the direction of greatest variance, as desired.

If truncated principal components regression is combined with centering, the model output becomes

$$\hat{y}(q) \;=\; \bar{b} + \sum_{i=1}^{n_\sigma} ((q - \bar{x})^{\mathrm{T}} \tilde{v}_i) \left(\frac{1}{\tilde{\sigma}_i} \right) (\tilde{u}_i^{\mathrm{T}} b), \tag{4.8}$$

where the number of components in the summation, n_σ, is a user specified parameter. Note that if the user specifies $n_\sigma = 0$, the model output is equal to the sample average of the target values for the k neighboring points stored in the vector b, given by Equation 4.3.2. At the other extreme where $n_\sigma = \min(k, n_d)$, this is equivalent to the OLS solution given by Equation 4.1.

4.3.3 Weighted Centering

In this section centering is generalized for the weighted least squares problem, $WA\nu = Wb$. To simplify notation, the matrix product $A_W \triangleq WA$ is decomposed as

$$\begin{aligned} A_W \;&=\; \begin{bmatrix} w_1 x_1^{\mathrm{T}} & w_1 \\ w_2 x_2^{\mathrm{T}} & w_2 \\ \vdots & \\ w_k x_k^{\mathrm{T}} & w_k \end{bmatrix}, \\ &=\; \begin{bmatrix} A_w & w \end{bmatrix}, \end{aligned}$$

and $b_W \triangleq Wb$, as before. The weighted analog of centering is to perform the singular value decomposition on the column space of A_w that is orthogonal to w,

$$\begin{aligned} \check{A}_w \;&\triangleq\; A_w \perp w, \\ &=\; A_w - w \frac{w^{\mathrm{T}} A_w}{w^{\mathrm{T}} w}. \end{aligned} \tag{4.9}$$

The row vector $\frac{w^{\mathrm{T}} A_w}{w^{\mathrm{T}} w}$ is the transpose of weighted sample average,

$$\begin{aligned} \bar{x}_w \;&\triangleq\; \frac{A_w^{\mathrm{T}} w}{w^{\mathrm{T}} w}, \\ &=\; \frac{\sum_{i=1}^k w_i^2 x_i}{\sum_{i=1}^k w_i^2}. \end{aligned}$$

If $w = \mathbf{1}_k$, this reduces to the ordinary sample average, $\bar{x}_w = \bar{x}$, where $\bar{x}$ is defined by Equation 4.6.

The matrix A_W can then be decomposed as

$$\begin{aligned} A_W &= \begin{bmatrix} A_w & w \end{bmatrix} \\ &= \begin{bmatrix} \check{A}_w & w \end{bmatrix} \begin{bmatrix} I & 0 \\ \bar{x}_w^{\mathrm{T}} & I \end{bmatrix}, \end{aligned}$$

and the weighted least squares problem can be restated as

$$\begin{aligned} A_W \nu &= b_W, \\ \begin{bmatrix} \check{A}_w & w \end{bmatrix} \begin{bmatrix} I & 0 \\ \bar{x}_w^{\mathrm{T}} & I \end{bmatrix} \nu &= b_W, \end{aligned}$$

or, equivalently,

$$\begin{bmatrix} \check{A}_w & w \end{bmatrix} \check{\nu} = b_W, \tag{4.10}$$

where

$$\check{\nu} \triangleq \begin{bmatrix} I & 0 \\ \bar{x}_w^{\mathrm{T}} & I \end{bmatrix} \nu. \tag{4.11}$$

This is a convenient transformation because the matrix has a trivial inverse,

$$\nu = \begin{bmatrix} I & 0 \\ -\bar{x}_w^{\mathrm{T}} & I \end{bmatrix} \check{\nu}. \tag{4.12}$$

Thus, once Equation 4.10 is solved for $\check{\nu}$, the model coefficients, ν, can be readily obtained from Equation 4.12.

The solution to the modified least squares problem, Equation 4.10, is

$$\check{\nu} = \left(\begin{bmatrix} \check{A}_w^{\mathrm{T}} \\ w^{\mathrm{T}} \end{bmatrix} \begin{bmatrix} \check{A}_w & w \end{bmatrix} \right)^{-1} \begin{bmatrix} \check{A}_w^{\mathrm{T}} \\ w^{\mathrm{T}} \end{bmatrix} b_W,$$

and since $w^{\mathrm{T}}\check{A}_w = 0$ (see Equation 4.9),

$$= \begin{bmatrix} \check{A}_w^{\mathrm{T}}\check{A}_w^{\mathrm{T}} & 0 \\ 0 & w^{\mathrm{T}}w \end{bmatrix}^{-1} \begin{bmatrix} \check{A}_w^{\mathrm{T}} b_W \\ w^{\mathrm{T}} b_W \end{bmatrix},$$
$$= \begin{bmatrix} \left(\check{A}_w^{\mathrm{T}}\check{A}_w^{\mathrm{T}}\right)^{-1} \check{A}_w^{\mathrm{T}} b_W \\ (w^{\mathrm{T}}w)^{-1} w^{\mathrm{T}} b_W \end{bmatrix}.$$

The last element in this matrix, $(w^{\mathrm{T}}w)^{-1} w^{\mathrm{T}} b_W$, is the weighted average of the target values. The first element in this matrix can be calculated by performing the singular value decomposition on the matrix $\check{A}_w = \check{U}\check{\Sigma}\check{V}^T$. Then the model output using weighted truncated principal components regression is given by,

$$\hat{y}(q) \;=\; \bar{b}_w + \sum_{i=1}^{n_\sigma} ((q - \bar{x}_w)^{\mathrm{T}} \check{v}_i) \left(\frac{1}{\check{\sigma}_i}\right) (\check{u}_i^{\mathrm{T}} b_W) \tag{4.13}$$

where n_σ is a user-specified parameter and $\bar{b}_w$ is the weighted sample average of b_W,

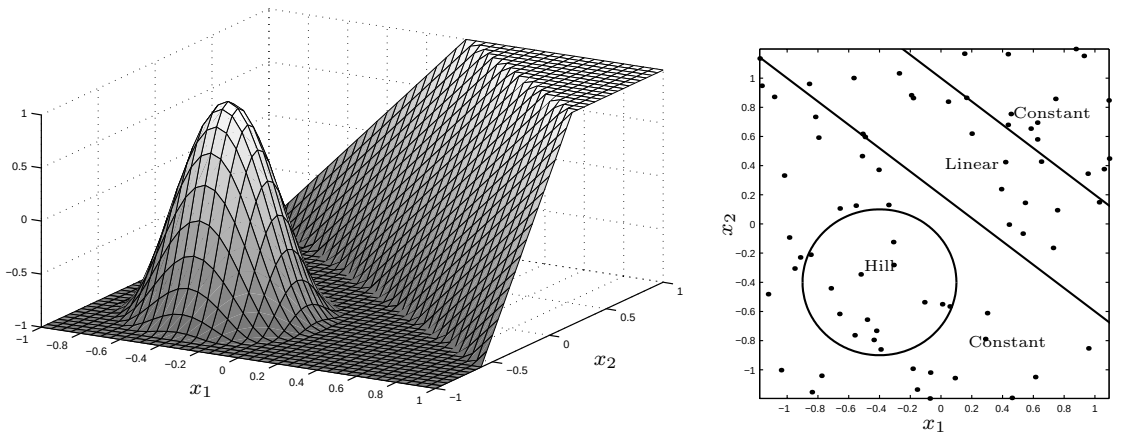
$$\bar{b}_w \triangleq \frac{\sum_{i=1}^{k} w_i^2 b_i}{\sum_{i=1}^{k} w_i^2}. \tag{4.14}$$

Parentheses are used in Equation 4.13 to emphasize that each element in the summation is a product of three scalars.

This generalization reduces to the solutions introduced previously when the parameters are adjusted appropriately. In the case where the user specifies $n_\sigma = 0$, the output is equal to the weighted local average, $\bar{b}_w$ given by Equation 4.14. If all of the weights are assigned a value of one, $w = \mathbf{1}_k$, the solution is equivalent to unweighted TPCR described by Equation 4.8. If $n_\sigma = \min(k, n_d)$ and $w = \mathbf{1}_k$, the solution is equivalent to the ordinary least squares solution described by Equation 4.1.

4.3.4 Example of Weighted TPCR

In this section weighted TPCR is applied to the RampHill function introduced in Chapter 2. Figure 4.4*a* shows the shape of the two-input function and Figure 4.4*b* shows a scatter plot of the seventy-five points used to build the local models. In this section, the data set is smaller than that used in Chapter 2 to introduce regions that naturally contain points that are nearly collinear.



(*a*) (*b*)

Figure 4.4: (*a*) Surface plot of the RampHill function. (*b*) Bird's-eye view of the sparse RampHill data set including labels of the significant features and a scatter plot of the data.

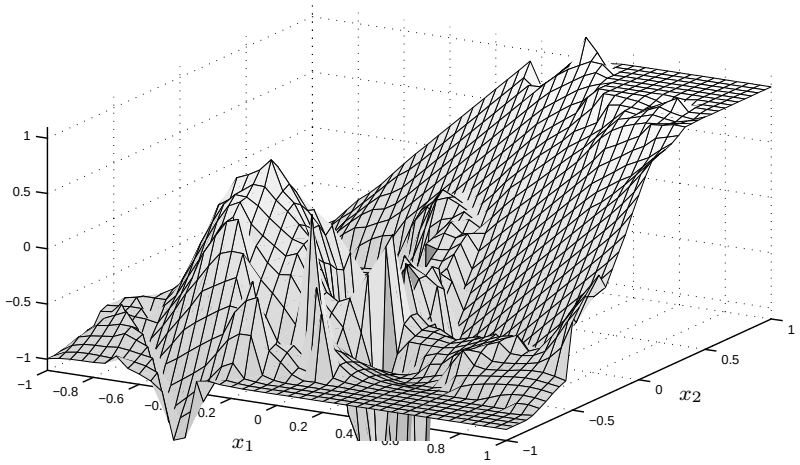
Figure 4.5 shows weighted TPCR applied to the sparse RampHill data set for $n_\sigma = 2$ and $n_\sigma = 1$. In the case of $n_\sigma = 2$, the regions with collinear neighbors have small singular values which causes the model output amplitude to be unreasonably large. For example, the model output becomes excessively negative near the base of the hill in Figure 4.5*a*.

In the case of $n_\sigma = 1$, the regions of collinearity do not cause excessive model outputs. However, the linear part of this function, the *ramp*, is no longer well approximated. This is because the subspace represented by the first principal component is rarely aligned with the direction of the ramp gradient. This represents a compromise between using ordinary least squares and a weighted average, which could be obtained by setting $n_\sigma = 0$.

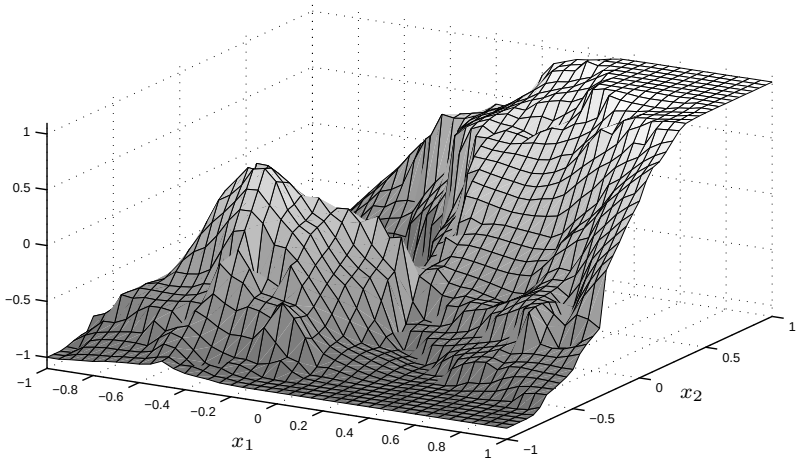
4.3.5 Soft Thresholding

Principal component threshold regression (PCTR) is another approach to regularization that incorporates singular value decomposition. Instead of truncating the summation to include only the n_σ largest singular values, PCTR multiplies each element in the sum by a threshold function [44]. The model output is given by,

$$\hat{y}(q) \;=\; \sum_{i=1}^{n_\sigma} (q^{\mathrm{T}} v_i) \left(\frac{f(\sigma_i)}{\sigma_i} \right) (u_i^{\mathrm{T}} b)\,, \tag{4.15}$$

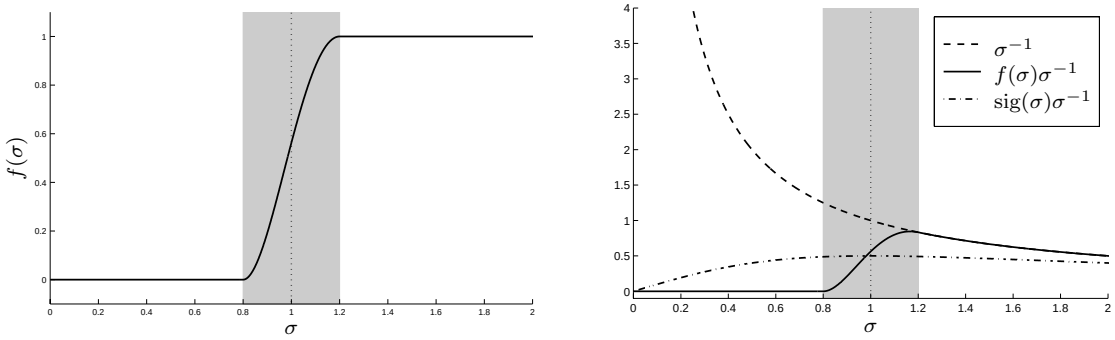


(*a*) $n_\sigma = 2$



(*b*) $n_\sigma = 1$

Figure 4.5: The output of a weighted TPCR for $n_\sigma = 2$ (*a*) and $n_\sigma = 1$ (*b*). The number of neighbors was set to $k = 5$ for both models.



(a) (b)

Figure 4.6: (a) The soft threshold function for $s_c = 1$ and $s_w = 0.2$. (b) Illustration of the soft threshold and sigmoidal functions multiplied by $\frac{1}{\sigma}$.

where $f(\cdot)$ is the threshold function. If weighted centering is used, the output is given by,

$$\hat{y}(q) \;=\; \bar{b}_w + \sum_{i=1}^{n_\sigma} ((q - \bar{x}_w)^{\mathrm{T}} \breve{v}_i) \left(\frac{f(\sigma_i)}{\breve{\sigma}_i} \right) (\breve{u}_i^{\mathrm{T}} b_W) \,.$$

Kugiumtzis *et al.* has suggested using a sigmoidal threshold function normalized by the residual variance for time series prediction [44]. Here a more general threshold function is proposed that has several advantages over the sigmoidal function. The new soft threshold is as follows:

$$f(\sigma) = \begin{cases} 0 & s_{\min} \leq \sigma, \\ \left(1 - \frac{(s_{\max} - \sigma)^2}{(s_{\max} - s_{\min})^2}\right)^2 & s_{\min} \leq \sigma < s_{\max}, \\ 1 & s_{\max} \leq \sigma, \end{cases} \tag{4.16}$$

where $s_{\min}$ and $s_{\max}$ are defined by

$$s_{\min} \;\triangleq\; s_c(1 - s_w), \tag{4.17}$$

$$s_{\max} \;\triangleq\; s_c(1 + s_w). \tag{4.18}$$

The user-specified parameters s_c and s_w represent the center and the percent width of the threshold function, respectively. This function is illustrated in Figure 4.6. The width of the threshold function is shown by the gray region and the center of the threshold is shown by the vertical dotted line.

The biweight function was used to generate a smooth transition from zero to one, hence the name *soft threshold*. The use of a smooth transition ensures that the model produces a smooth output. Since the threshold function is exactly equal to one for large values of σ, it does not introduce any bias in these components of the summation (Equation 4.15). Similarly, since the threshold function is exactly zero for small values of σ, these components of the summation are eliminated altogether. In contrast the sigmoid function proposed by Kugiumtzis *et al.* introduces bias for all values of σ and does not completely eliminate any elements of the summation, as shown by Figure 4.6*b*.

Since the expected value of each singular value squared is proportional to the number of rows in the A matrix, $\sigma_i^2 \propto k \quad$ for all i, it is good practice to normalize the singular values before applying the threshold function. Thus, $f(\frac{\sigma_i}{\sqrt{k}})$ was used for all of the results reported here, instead of $f(\sigma_i)$.

Example of Soft Thresholds for PCTR

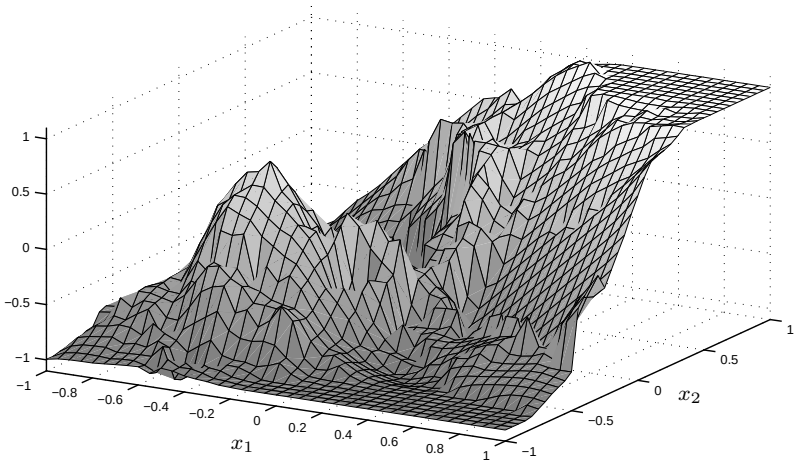
Figure 4.7 shows the result of applying PCTR to the sparse RampHill data set for two different values of s_c. In both cases the large outputs caused by collinearity were eliminated. For the smaller value, $s_c = 0.03$, much of the linear region is accurately modeled because both principal directions were used to construct the model. As s_c is increased, the model output becomes more similar to weighted averaging, as shown by Figure 4.7*b*.

4.4 Tradeoffs

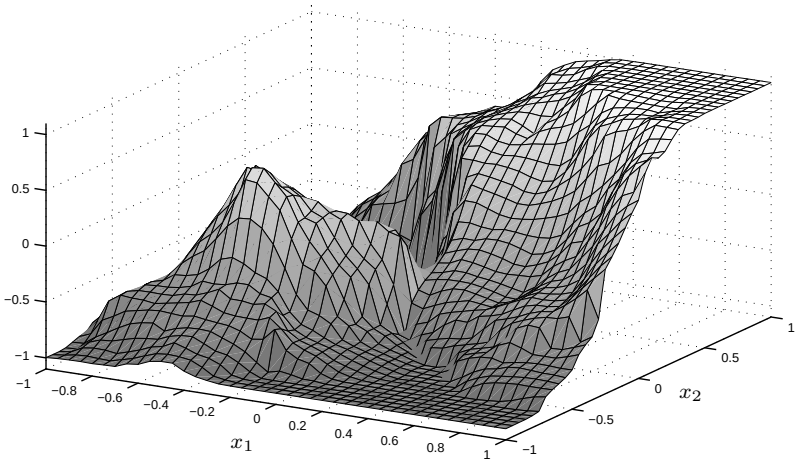
As with most decisions, there are tradeoffs between each method of regularization that the user has to choose from. This section lists some of the tradeoffs between the two new types of regularization described in this chapter.

4.4.1 Influence of Input Variables

Vectored ridge regression (VRR) enables the user to explicitly specify the tradeoff between minimizing the weighted sum of squared errors and decreasing the influence of each model input variable. This is especially useful if the user knows some inputs are particularly noisy or weakly related to the model output. The various forms of principal components regression (PCR) do not have this property.



(*a*) $s_c = 0.03$



(*b*) $s_c = 0.10$

Figure 4.7: The output of a weighted PCTR using the soft threshold for two values of s_c. The threshold width was fixed at $s_w = 0.20$ and both elements of the sum were used, $n_\sigma = 2$, for both cases.

4.4.2 Direct Control of Model Subspace

If the user has reason to believe that the points lie in a subspace of the input domain, this can specifically be embedded in the model with truncated PCR, which effectively limits the model subspace. VRR does not have this feature.

4.4.3 Adaptive Regularization

The use of a threshold function in PCR allows the model to adapt to the degree of collinearity in each neighborhood. If a neighborhood lies on a low dimensional plane, the corresponding principal directions with near-zero singular values are automatically ignored. If the points thoroughly span the input domain and the matrix product is well conditioned, the solution is the same as the least squares solution.

VRR imposes the same penalty regardless of the distribution of the points or the conditioning of the matrix product $A^{\mathrm{T}}A$.

4.4.4 Number of Parameters

VRR has n_d+1 ridge parameters, but since $r_{n_d+1} = 0$, the user only has to specify n_d values. The additional parameters, as compared to ordinary least squares, are an advantage if the user knows which inputs are the most important, but this knowledge may be difficult to quantify appropriately. If the user does not possess this type of knowledge, the task of specifying n_d ridge parameters may be overwhelming.

Truncated PCR with soft thresholding requires the user to specify three parameters: the number of principal directions to use for the model output, the center of the threshold function, and the relative width of the threshold transition region. Even if the user is intimately familiar with the process from which the data was collected, the appropriate values for these parameters will rarely be known. However, since there are only three parameters that need to be specified, it may be reasonable to use the method of Futz[5] to find good values.

The task of choosing appropriate parameter values is discussed more fully in Chapter 5.

4.4.5 Gradient Calculation

A key advantage of ridge regression is that it enables efficient calculation of the model output gradient with respect to the ridge parameters and with respect to the penalty weights.

[5]Also known as "trial and error," or "futz around" with it.

These gradients are derived in Appendix A and they play an important role in parameter optimization discussed in Chapter 5.

The author is unaware of any method for efficiently calculating the gradient of the model output with respect to the penalty weights or regularization parameters for PCR. The gradient may be estimated by dithering, but this approach requires the model to be reconstructed for each element of the gradient and is impractical for large values of k, the dimension of the penalty weight vector. This is discussed more fully in Chapter 5.

4.4.6 Computational Efficiency

Cholesky decomposition can be used to efficiently solve the normal equations for VRR [118, p. 96–8]. Singular value decomposition, employed by the various forms of PCR, requires about an order of magnitude more computation [111, p. 263].

This does not necessarily mean that the total time required to calculate a model output will be much less using VRR. Often the majority of computation is used to find the $k+1$ nearest neighbors in the data set and the relative importance of efficient model construction is minimal. This varies greatly depending on many factors including the data set distribution, the data set size, and the number of neighboring points that must be found.

4.5 Summary

This chapter introduced two new methods of regularization for local linear models. Both methods reduce the effects of collinearity, a problem that plagues local linear models due to the small neighborhoods used to construct the models.

Both methods give the user greater control than their precedents. Vectored ridge regression (VRR) enables the user to reduce the influence of each model input. Principal components threshold regression (PCTR) allows the user to specify how ill-conditioned the data matrix may become before corrective measures are taken.

This chapter also described a generalization of centering, a common method applied to principal components analysis to ensure that the principal directions correspond to the directions of greatest covariance in the input domain. Centering was generalized to the case of weighted sum of squared errors.

Each of the methods introduced in this chapter requires that the user specify additional parameter values. For vectored ridge regression the user must specify the ridge parameter values and for the new method of principal components regression (PCR) the user must

specify the number of singular values, the threshold center, and the threshold transition width. In practice, the user will rarely have an intuition for how to choose good values for these parameters. The next chapter addresses this problem.

Chapter 5

Local Model Optimization

Everything that can be invented has been invented.
—Commissioner, U.S. Office of Patents, 1899.

One of the most vexing problems facing users who wish to construct a local model is how to choose appropriate values for the model parameters. Since the best parameter values depend on the properties of the data set, there is little to guide users in making this decision.

This chapter introduces a method for optimizing the parameter values to minimize the average model error. The advantages of this method are that it relieves the user of the burden of specifying critical parameter values, it gives the user control of the computation used for optimization, and it improves the model accuracy starting with the initial values provided by the user.

This chapter is organized as follows. Section 5.1 introduces iterative optimization algorithms. Section 5.2 discusses how to apply these methods to optimize the metric for local averaging. Sections 5.3 and 5.4 discussed optimization of vectored ridge regression (VRR) and principal components regression (PCR), respectively. Section 5.5 introduces cyclic coordinate optimization and a generalization of this technique. Sections 5.6 and 5.7 give complete optimization algorithms for vectored ridge regression and principal components regression, respectively. Section 5.8 discusses some of the issues encountered in practice with these algorithms.

5.1 Iterative Optimization Algorithms

There are many optimization algorithms that can minimize a user-defined penalty function. For local models the natural choice for the penalty function is the leave-one-out cross validation error (CVE)[1].

If the gradient of the penalty function with respect to the model parameters can be calculated, there are a number of iterative optimization algorithms to choose from [43, 119]. Algorithms also exist for cases where the gradient is not available, but they are less efficient [43].

Most of the gradient-based algorithms consist of two steps that are repeated until the algorithm meets a user-specified convergence criteria. During the first step a direction of descent is calculated, g. During the second step, a line search is used to determine how much the model parameters should be adjusted in the direction g. Specifically, if the model parameters are contained in a vector γ, the line search finds a value for the scalar α that minimizes $\mathrm{P}(\gamma + \alpha g)$, where $\mathrm{P}(\cdot)$ is the user-specified penalty function.

For the results reported here, the parallel tangents algorithm (PARTAN) was used to generate the directions of descent and the golden section method was used for each line search [43, 119]. PARTAN was chosen because it is one of the most stable conjugate gradient methods and it can theoretically find the minimum of a quadratic penalty function in n_γ steps, where n_γ is the number of parameters [119, p. 240]. Since most continuous penalty functions are accurately approximated by a quadratic function near local minima, the convergence of PARTAN is typically much faster than gradient descent.

Although all conjugate gradient algorithms converge very quickly on quadratic problems, the performance of these algorithms varies on non-quadratic problems and the rate of convergence can be substantially decreased when inexact line search algorithms are used. PARTAN was also chosen because it is among the least sensitive algorithms to inexact line searches. A detailed description of this algorithm and its properties is given by Luenberger [119] and Bazaraa *et al.* [43].

During parameter optimization, the line search algorithm usually requires many more evaluations of the penalty function than the optimization algorithm. If the computational cost of evaluating the penalty gradient, $\nabla_\gamma \mathrm{P}(\gamma)$, is significantly larger than evaluating the penalty itself, $\mathrm{P}(\gamma)$, it may be more efficient to use a line search that does not require

[1] Friedman has noted that leave-one-out cross-validation is not always the best choice, but he did not suggest a better alternative [33].

the gradient. Golden section was selected as the line search algorithm because it is among the fastest[2] line search algorithms that does not require the gradient. Golden section also allows the line search to be stopped after the region of uncertainty has been reduced by a user-specified amount, a technique that was used here.

The optimization algorithm and line search method were chosen carefully for their fast convergence and suitability to the type of penalty function used. It is important to note, however, that any of the other gradient-based optimization algorithms and line search methods could have been used to solve the problems described in this chapter.

5.2 Local Averaging Metric Optimization

Choosing an appropriate measure of distance, or metric[3], for local models is an important decision that is often overlooked. The Euclidean distance,

$$\begin{aligned} \mathrm{D_E}\,(q, x_i)^2 &= \sum_{j=1}^{n_d} (q_j - x_{i,j})^2, \\ &= (q - x_i)^{\mathrm{T}} (q - x_i), \end{aligned} \tag{5.1}$$

where q is the query vector and x_i is the ith nearest neighbor, is the most common choice. But, there is no reason to believe this is the best choice in general; researchers pick this metric because of its simplicity and intuitive geometric appeal.

Choosing a more general measure with more parameters can drastically affect model accuracy [26, 120–122]. For example, a weighted Euclidean distance,

$$\mathrm{D_{WE}}(q, x_i)^2 = (q - x_i)^{\mathrm{T}} \Lambda^2 (q - x_i),$$

could be used where $\Lambda \in \mathbb{R}^{m \times m}$ is any positive semidefinite matrix. However, data sets are often too small to estimate n_d^2 parameter values and an optimization over so many

[2]Fibonacci line search is slightly faster, but it requires that number of iterations per a line search be fixed [43, p. 275].

[3]In this chapter *metric* means a measure of nearness between two points. It may not have the usual metric properties.

parameters is computationally impractical for larger data sets. For this work the diagonally weighted Euclidean metric,

$$\begin{aligned} \mathrm{D}_{\mathrm{WE}}(q, x_i)^2 &= \sum_{j=1}^{n_d} \lambda_j^2 (q_j - x_{i,j})^2, \\ &= (q - x_i)^{\mathrm{T}} \Lambda^2 (q - x_i), \end{aligned} \tag{5.2}$$

where $\Lambda = \mathrm{diag}(\lambda)$ and $\lambda \in \mathbb{R}^{n_d}$, was chosen. This metric is much more flexible than the Euclidean metric and yet it has few enough parameters that optimization is possible. This metric also has the ability to ignore irrelevant input variables and increase the influence of the most important input variables, much like vectored ridge regression (VRR). In Chapter 6 a more constrained form of this metric will be introduced that is tailored to the time series prediction problem.

5.2.1 Metric Gradient

The gradient of the cross-validation error (CVE) with respect to the metric parameters λ must be calculated in order to optimize them with an iterative optimization algorithm. The average CVE is defined as,

$$\mathrm{CVE}(\lambda) \triangleq \sum_{i=1}^{n_c} p\left(y_{c(i)} - \hat{y}^-(x_{c(i)})\right), \tag{5.3}$$

where n_c controls the accuracy of the estimated error, $c(i)$ is the data set index of the ith cross-validation point and $p(\cdot)$ is a user-specified penalty function[4]. The local model output, $\hat{y}^-(x_{c(i)})$, is from a model constructed with the $c(i)$th point omitted from the data set and then used as the model input.

The gradient of the CVE is given by,

$$\nabla_\lambda \mathrm{CVE}(\lambda) = -\sum_{i=1}^{n_c} \frac{dp(\varepsilon)}{d\varepsilon} \nabla_\lambda \hat{y}^-(x_{c(i)}),$$

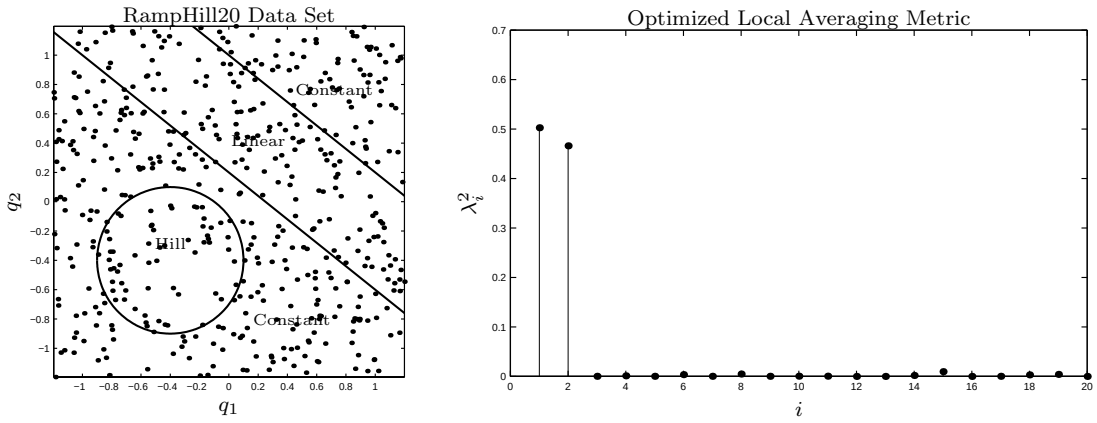
where

$$\varepsilon \triangleq y_{c(i)} - \hat{y}^-(x_{c(i)}).$$

[4]The squared error penalty function, $p(\varepsilon) = \varepsilon^2$, is the most common choice, but any smooth function could be used.

The derivation of the model output gradient with respect to the metric parameters, $\nabla_\lambda \hat{y}(\cdot)$, for VRR is given in Appendix A along with an algorithm that describes how to calculate this gradient efficiently.

5.2.2 Weighted Averaging Example

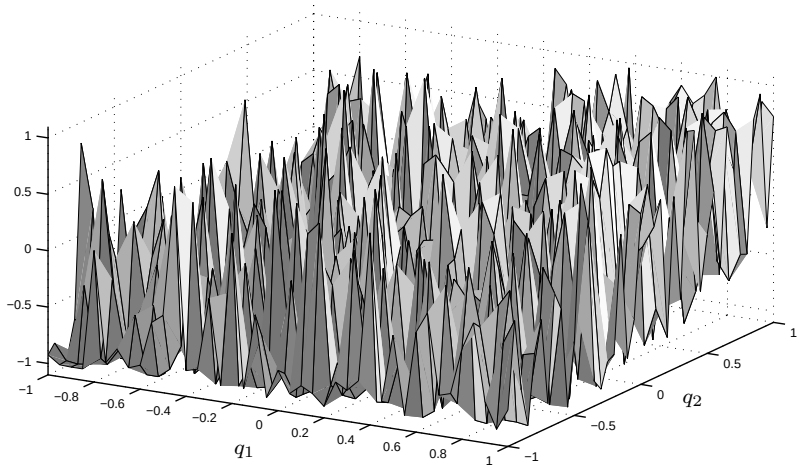


(a) (b)

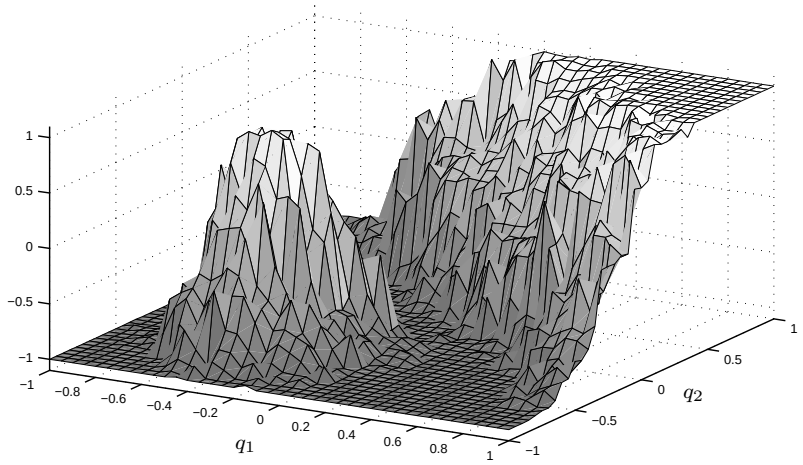
Figure 5.1: (a) Scatter plot of the two relevant inputs for the RampHill20 data set. The data set also contained 18 spurious inputs that are not shown. (b) The local averaging metric parameters, λ^2, after optimization. The number of neighbors was fixed at $k = 5$. The metric has been normalized so that $\lambda^{\mathrm{T}}\lambda = 1$.

Once the gradient is available, the metric parameters can be optimized using the iterative optimization algorithm discussed in Section 5.1. To illustrate the benefit of metric optimization, the metric parameters were optimized for the RampHill function described in Chapter 2. In this section the data set, RampHill20, contains 250 points and has eighteen additional model inputs with the same random distribution as the original two model inputs, but they have no effect on the process output. A scatter plot of the relevant inputs is shown in Figure 5.1a.

The optimization began with all of the metric weights equal to one, which is equivalent to the Euclidean metric, and converged to the values shown in Figure 5.1b. Since the process output only depends on the first two inputs, the metric parameters for the eighteen spurious inputs became nearly zero after optimization, as shown by Figure 5.1b. Thus the optimization was almost able to eliminate the influence of the spurious inputs and produce a model that is equivalent to one constructed with just the first two model inputs.



(a) Euclidean Metric



(b) Optimized Weighted Euclidean

Figure 5.2: These figures illustrate the potential benefit of metric optimization applied to local averaging. Figure (a) shows the local averaging output using the Euclidean metric and (b) shows the local averaging output after metric optimization. The data set contained 250 points with 18 spurious model inputs. Both models used $k = 5$ nearest neighbors to calculate the local average.

Figure 5.2*a* shows the local averaging model output for the RampHill20 data set using the Euclidean metric and Figure 5.2*b* shows the model output for the weighted Euclidean metric after optimization[5]. Although the local averaging output is a smooth function of the twenty model inputs in both cases, these figures appear noisy because of the effect of the spurious inputs. If the optimization had made the spurious-inputs' metric weights exactly zero, the optimized output would have been a smooth function of the two relevant inputs.

5.2.3 Local Minima

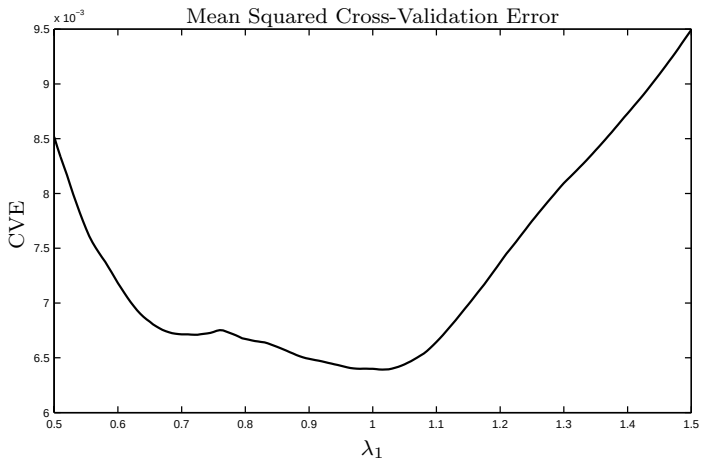


Figure 5.3: Illustration of multiple local minima as a function of the metric parameters.

A disadvantage of iterative optimization is that it only finds the nearest local minimum, which is rarely the global minimum. As an example of this, consider the the RampHill20 data set. If only the two relevant variables are used to construct the model, there are only two metric parameters, λ_1 and λ_2. Since the cross-validation error is invariant to the scale of the metric parameters, $\mathrm{CVE}(\lambda) = \mathrm{CVE}(\alpha\lambda)$ for all $\alpha > 0$, there is only one degree of freedom. Figure 5.3 shows a plot of the cross-validation mean squared error as a function of λ_1 for $\lambda_2 = 1$. In this case there are two local minima, one at approximately $\lambda_1 = 0.7$ and a global minimum at approximately $\lambda_1 = 1.02$.

Although local minima prevent a global minimization, optimization of local averaging models usually improves the model accuracy substantially.

[5]The optimized model output is considerably better than examples in previous chapters because the RampHill20 data set contains more points.

5.3 Vectored Ridge Regression

If a local linear model is used with vectored ridge regression (VRR), the ridge parameters can be optimized jointly with the metric parameters. Let γ be defined as the concatenation of the metric parameters, λ, and the ridge parameters, r, so that $\gamma^{\mathrm{T}} \triangleq [\lambda^{\mathrm{T}} \;\; r^{\mathrm{T}}]$. The gradient of the CVE with respect to γ is given by,

$$\nabla_\gamma \mathrm{CVE}(\gamma) \;=\; -\sum_{i=1}^{n_c} \frac{dp(\varepsilon)}{d\varepsilon} \nabla_\gamma \hat{y}^-(x_{c(i)}),$$

where $p(\cdot)$ is the penalty function, $\varepsilon = y_{c(i)} - \hat{y}^-(x_{c(i)})$, and the gradient of the model output is given by,

$$\nabla_\gamma \hat{y}^{-i}(x_{c(i)}) = \begin{bmatrix} \nabla_\lambda \hat{y}^{-i}(x_{c(i)}) \\ \nabla_r \hat{y}^{-i}(x_{c(i)}) \end{bmatrix}. \tag{5.4}$$

Both of these gradients are derived in Appendix A and can be calculated efficiently once the model has been constructed.

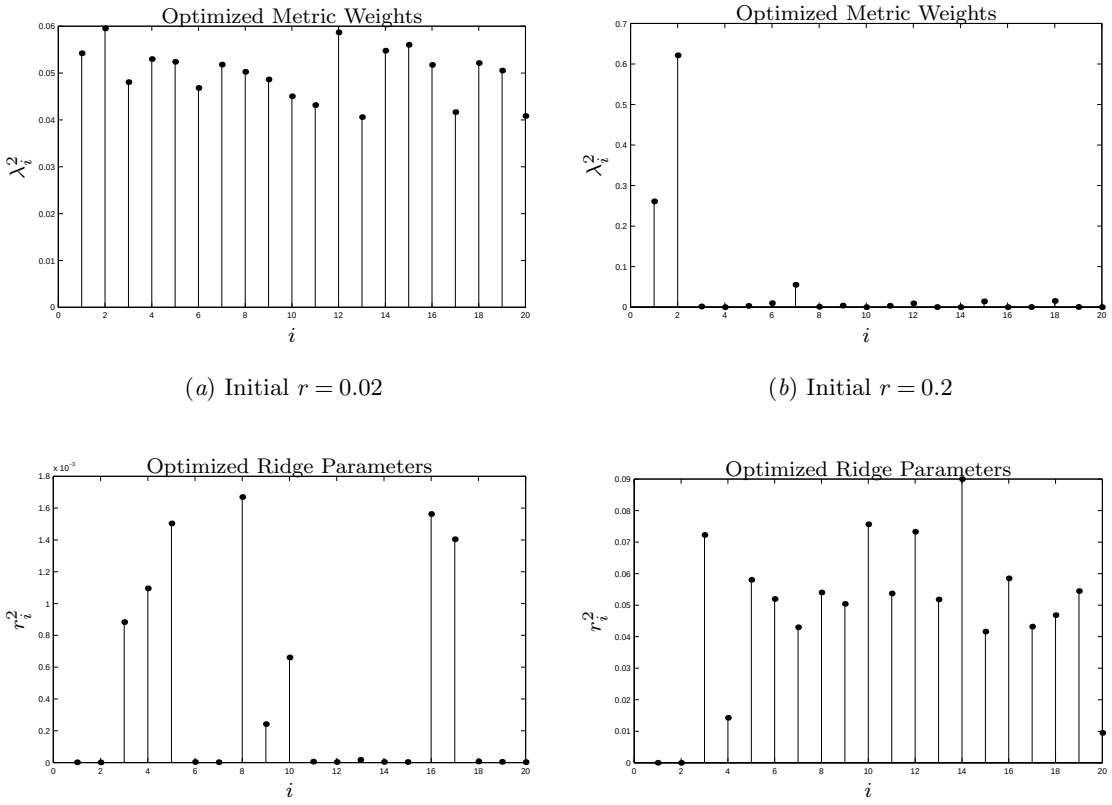
5.3.1 Example

Optimization of local linear models is much more sensitive to the choice of initial parameter values than local averaging. To illustrate this the metric and ridge parameters were jointly optimized for the RampHill20 data set. Figure 5.4 illustrates the parameter values after optimization with two different values for the ridge parameters. Since the process output is only a function of the first two variables, the optimized metric weights should be small for the last eighteen input variables and the optimized ridge parameters should be large for these variables. As shown by the figure, this was only achieved when the ridge parameters were initialized with the relatively large value of $r = 0.2$.

Figure 5.5 shows the model output after optimization for each case and confirms that the larger initial ridge parameters produced a better model after optimization.

5.4 Principal Components Regression

Since there is no known means of calculating the gradient of the model output with respect to the singular values of the A matrix, it is not possible to use a gradient-based optimization algorithm for principal components regression (PCR). Although it is possible to estimate



(a) Initial $r = 0.02$

(b) Initial $r = 0.2$

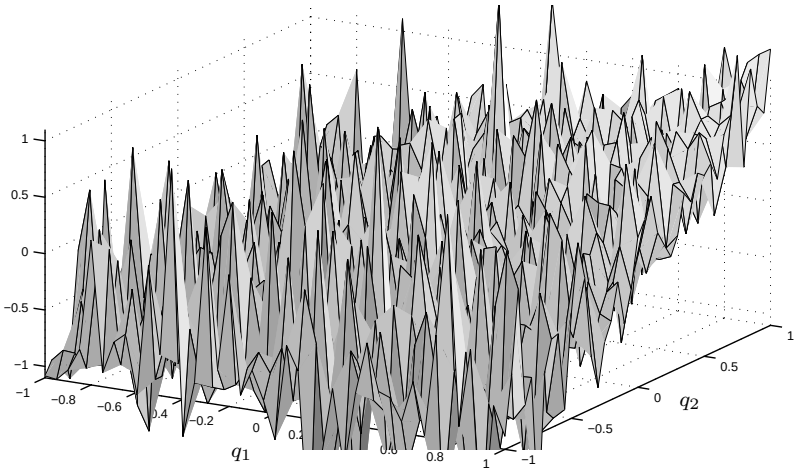
(c) Initial $r = 0.02$

(d) Initial $r = 0.2$

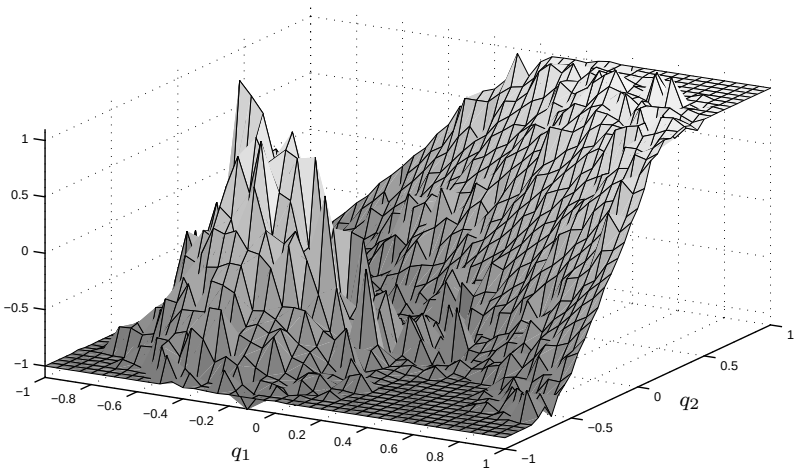
Figure 5.4: Illustration of the sensitivity to initial parameters for vectored ridge regression. Figures (a) and (b) show the metric parameters after optimization with initial ridge parameters $r = 0.02$ and $r = 0.2$, respectively. Similarly, Figures (c) and (d) show the ridge parameters after optimization with $r = 0.02$ and $r = 0.2$, respectively. In both cases the metric parameters were initialized with $\lambda = 1$, which is equivalent to the Euclidean metric.

the gradient by dithering the parameter values, this approach is computationally intensive because it requires that the model be constructed at least $n_\gamma + 1$ times to estimate the gradient at a single point, where n_γ is the number of parameters being optimized [123, pp. 66-93]. Optimization algorithms also exist that do not require the calculation of the gradient, but they are much slower than gradient based methods [43].

In the following section a new approach is taken to jointly optimize principal components regression (PCR) model parameters.



(a) Initial $r = 0.02$



(b) Initial $r = 0.2$

Figure 5.5: Local linear model output after optimization with two different initial values for the ridge parameters. (a) The initial value for the ridge parameters was too small and the optimization became trapped in a shallow local minima. (b) A good initial value was chosen for the ridge parameters and the final model is much more accurate.

5.5 Cyclic Coordinate Optimization

The gradient-based optimization described in the previous sections can greatly improve the initial metric and ridge parameter values provided by the user. However, this approach can not be used to optimize some of the other model parameters. For example, since the neighborhood size is an integer, gradient-based methods can not be used to optimize this parameter. Similarly, the center of the soft threshold function for PCR can not be optimized because there is no means of directly calculating the partial derivative of the model output with respect to this parameter.

To optimize these parameters, an algorithm that does not require the gradient must be used. One of the simplest of these algorithms is the cyclic coordinate method. This method optimizes each parameter one at a time, and then repeats until convergence. [43, pp. 283–5]. For example, if the parameters to be optimized are stored in a vector $\gamma \in \mathbb{R}^{n_\gamma}$, the cyclic coordinate method is as follows.

Algorithm 5.1: Cyclic Coordinate Method

1. For $i = 1$ to n_γ,
   1.1 $\gamma_i := \underset{\alpha}{\operatorname{argmin}}\, \mathrm{CVE}([\gamma_1, \ldots, \gamma_{i-1}, \alpha, \gamma_{i+1}, \ldots, \gamma_{n_\gamma}]^{\mathrm{T}})$.
   1.2 Next i.
2. If not converged, then Goto 1.

Since each step in the loop can only decrease the cross-validation error, this method can only improve the model performance and under very general conditions the algorithm is guaranteed to converge [43, p. 285].

5.5.1 Semi-global Line Search

Each step in the inner loop of the cyclic coordinate method tries to find the value of a single parameter that minimizes the cross-validation error. Since only one parameter is optimized at each step, this is essentially a one-dimensional minimization problem, also known as the line search problem.

If the parameter to be optimized is an integer, a user-specified range of values can be searched for the best value. For example, the number of neighbors, k, could be optimized by calculating the cross-validation error (CVE) for a range of values, $\{k_{\min}, k_{\min}+1, \ldots, k_{\max}\}$, and retaining the value with the smallest CVE.

If the parameter is a real number, any of a number of line search algorithms could be used to find a local minimum. However, a semi-global line search algorithm is preferable

if the CVE contains many shallow local minima, as is case with most of the local model parameters.

The semi-global line search algorithm used here tries increasing and decreasing the parameter value by a range of amplification factors, a set of scalar multipliers. For example, if γ_i is the parameter being minimized and Φ is a set of possible amplification factors, the minimization in step 1.1 would consist of evaluating the CVE with $\alpha = \phi\gamma_i$ for each $\phi \in \Phi$. The parameter γ_i would then be replaced with the value that minimized the CVE.

To ensure that a wide range of parameter values is examined, the amplification factors were chosen to be evenly spaced on a logarithmic scale. For example, if the user wished to evaluate the CVE at only eleven points and wanted to try amplification factors ranging from $\frac{1}{10}$ to 10, the amplification factors would be {0.100, 0.158, 0.251, 0.398, 0.631, 1.00, 1.58, 2.51, 3.98, 6.31, 10.0}.

This approach requires the user to specify three parameters: the number of amplification factors, n_ϕ, the maximum amplification factor, $\phi_{\max}$, and the minimum amplification factor, $\phi_{\min}$. For all of the results reported here $\phi_{\min} = \frac{1}{\phi_{\max}}$. The remaining two user-specified parameter values selected for the results reported here are given in Table 5.1 in Section 5.5.2.

After the cyclic coordinate method converges, the parameter values can be found with greater precision by reducing the range of the amplification factors. For example, after initial convergence, the range could be reduced to $\phi_{\min} = 0.2$ and $\phi_{\max} = 5$. This will be discussed further in the next section.

5.5.2 Generalized Cyclic Coordinate Method

The cyclic coordinate method's rate of convergence is much slower than gradient-based optimization methods. If the gradient can be calculated for only some of the model parameters, the rate of convergence can be increased by combining the cyclic coordinate method with a gradient-based algorithm such as PARTAN.

Here a generalization of the cyclic coordinate method is described that combines cyclic optimization with gradient-based optimization. During each cycle, the gradient parameters are jointly optimized for a user-specified number of steps. The remaining parameters are found by exhaustive search for integers and by semi-global line search for real-valued parameters. This approach is substantially faster than using the cyclic method to optimize each of the parameters one at a time.

To illustrate this method, suppose the gradient of the CVE can be calculated for the parameters, $\{\gamma_1, \ldots, \gamma_{n_g}\}$, and not for the rest, $\{\gamma_{n_g+1}, \ldots, \gamma_{n_\gamma}\}$. In this case the CVE may be written as a function of two vectors, $\mathrm{CVE}(\tilde{\gamma}, \acute{\gamma})$, where $\tilde{\gamma} \triangleq [\gamma_1, \ldots, \gamma_{n_g}]^{\mathrm{T}}$ and $\acute{\gamma} \triangleq [\gamma_{n_g+1}, \ldots, \gamma_{n_\gamma}]^{\mathrm{T}}$. Using this notation, the generalized cyclic coordinate method is given as follows.

Algorithm 5.2: Generalized Cyclic Coordinate Method

1. Define $\tilde{\gamma} \triangleq [\gamma_1, \ldots, \gamma_{n_g}]^{\mathrm{T}}$ as a vector that contains the parameters for which the gradient may be calculated and $\acute{\gamma} \triangleq [\gamma_{n_g+1}, \ldots, \gamma_{n_\gamma}]^{\mathrm{T}}$ as a vector that contains the remaining parameters. Define $\Phi = \{\frac{1}{\phi_{\max}}, \ldots, \phi_{\max}\}$ as a finite set of real-valued amplification factors.
2. Initialize the stopping criteria:
   $n_i := 0$,
   $n_r := 0$.
3. Store the current value of CVE:
   $\mathrm{CVE}_{\mathrm{prev}} := \mathrm{CVE}(\tilde{\gamma}, \acute{\gamma})$.
4. Perform Cyclic Optimization.
   For $i = 1$ to $(n_\gamma - n_g)$,
   4.1 If $\acute{\gamma}_i$ is integer valued, then perform an exhaustive search:
   $\acute{\gamma}_i := \underset{\gamma_{i,\min} \le \alpha \le \gamma_{i,\max}}{\operatorname{argmin}} \mathrm{CVE}(\tilde{\gamma}, [\acute{\gamma}_1, \ldots, \acute{\gamma}_{i-1}, \alpha, \acute{\gamma}_{i+1}, \ldots, \acute{\gamma}_{n_{\acute{\gamma}}}]^{\mathrm{T}})$.
   4.2 Else, perform a semi-global line search,
   $\phi^\star := \underset{\phi \in \Phi}{\operatorname{argmin}} \mathrm{CVE}(\tilde{\gamma}, [\acute{\gamma}_1, \ldots, \acute{\gamma}_{i-1}, \phi\acute{\gamma}_i, \acute{\gamma}_{i+1}, \ldots, \acute{\gamma}_{n_{\acute{\gamma}}}]^{\mathrm{T}})$,
   and update the parameter value,
   $\acute{\gamma}_i := \phi^\star \acute{\gamma}_i$.
   4.3 Next i.
5. Perform gradient-based optimization:
   For $i = 1$ to n_u,
   5.1 Calculate $\nabla_{\tilde{\gamma}} \mathrm{CVE}(\tilde{\gamma}, \acute{\gamma})$.
   5.2 Calculate a new direction of descent, g.
   5.3 Perform line search:
   $\alpha := \underset{\alpha \ge 0}{\operatorname{argmin}} \mathrm{CVE}(\tilde{\gamma} + \alpha g, \acute{\gamma})$.
   5.4 Update the parameters:
   $\tilde{\gamma} := \tilde{\gamma} + \alpha g$.
   5.5 Next i.
6. Update the count of iterations:
   $n_i := n_i + 1$.
7. If $n_i = n_{i,\max}$, then the number of cycles has reached the user-specified limit. Exit function.
8. If $\mathrm{CVE}(\tilde{\gamma}, \acute{\gamma}) \neq \mathrm{CVE}_{\mathrm{prev}}$, then the optimization has not yet converged. Goto 3.

9. If $n_r = n_{r,\max}$, then the semi-global range has been reduced the user-specified number of times. Exit function.

10. Reduce the range for the semi-global line search:
$\phi_{\max} := 1 + (\phi_{\max} - 1)/2$,
recalculate the set of amplification factors,
$\Phi = \{\frac{1}{\phi_{\max}}, \ldots, \phi_{\max}\}$,
and increment the count of reductions:
$n_r := n_r + 1$.

11. Goto 3.

This algorithm uses two stopping criteria. First, if the number of cycles exceeds the user-specified limit, $n_{i,\max}$, the algorithm stops and returns the best parameter values that were found. Second, the algorithm will stop if the CVE did not change during a cycle and the range of the semi-global line search has been reduced $n_{r,\max}$ times.

If an ideal line search were used the second criteria would never be met because the line search in Step 5.3 would always reduce the CVE slightly since g is a direction of descent[6]. However, a practical implementation of a line search constrained to find the approximate local minimum in a limited number of steps may return a step size of $\alpha = 0$, which would leave the CVE unchanged.

Parameter	Default	Description
$n_{i,\max}$	7	The maximum number of cycles that are allowed before the optimization is stopped.
n_u	5	The number of gradient-based updates per a cycle.
$\phi_{\max}$	10	The maximum value of the amplification factors.
n_ϕ	11	The number of amplification factors considered in each semi-global line search.
$n_{r,\max}$	4	The maximum number of reductions in the range of amplification factors.

Table 5.1: All results reported here were generated with these user-specified parameter values for the generalized cyclic optimization method.

Table 5.1 summarizes the user-specified parameters for the generalized cyclic coordinate method. Each of these parameters controls the tradeoff between the final model accuracy and the time allowed to optimize the parameters. This is an important advantage of this approach because it relieves the user of responsibility for picking the model parameter values and allows control of the time required to build the model instead.

[6]Theoretically, the line search will not change the parameters if an exact local minimum is found and the magnitude of the gradient is zero, but this does not occur in practice.

In the following sections the generalized coordinate method is used to optimize vectored ridge regression (VRR) and principal components regression (PCR).

5.6 Vectored Ridge Regression Optimization

Using the generalized cyclic coordinate method, the number of neighbors may be jointly optimized with the ridge parameters and the metric parameters. Here the cross-validation error is denoted as $\mathrm{CVE}(k,\gamma)$ where k is the number of neighbors and $\gamma^{\mathrm{T}} \triangleq [\lambda^{\mathrm{T}} \;\; r^{\mathrm{T}}]$, where $\lambda \in \mathbb{R}^{n_d}$ is a vector of the metric parameters and $r \in \mathbb{R}^{n_d}$ is a vector of the ridge parameters. The complete algorithm is given below.

Algorithm 5.3: Weighted Vectored Ridge Regression Optimization

1. Define $\gamma^{\mathrm{T}} \triangleq [\lambda^{\mathrm{T}} \;\; r^{\mathrm{T}}]$ where λ are the metric parameters and r are the ridge parameters. Define k as the number of neighbors used to build the model.
2. Initialize the stopping criteria:
   $n_i := 0$,
   $\mathrm{CVE}_{\mathrm{prev}} := \infty$.
3. Optimize the number of neighbors by an exhaustive line search:
   $k := \underset{k_{\min} \le i \le k_{\max}}{\operatorname{argmin}} \; \mathrm{CVE}(i,\gamma)$.
4. For $i = 1$ to n_u,
   4.1 Calculate the gradient:
   $\nabla_\gamma \mathrm{CVE}(k,\gamma)$.
   4.2 Calculate a new direction of descent, g.
   4.3 Perform a line search:
   $\alpha := \underset{\alpha \ge 0}{\operatorname{argmin}} \; \mathrm{CVE}(k,\gamma + \alpha g)$.
   4.4 Update the parameters:
   $\gamma := \gamma + \alpha g$.
   4.5 Next i.
5. If $n_i = n_{i,\max}$, then the number of cycles has reached the user-specified limit. Exit function.
6. If $\mathrm{CVE}(k,\gamma) = \mathrm{CVE}_{\mathrm{prev}}$, then the optimization has converged. Exit function.
7. Update the stopping criteria:
   $n_i := n_i + 1$,
   $\mathrm{CVE}_{\mathrm{prev}} := \mathrm{CVE}(k,\gamma)$.
8. Goto 3.

Here the new user-specified parameters are the initial values for the metric weights λ, the initial values for the ridge parameters r, and the range of values for k, $k_{\min}$ and $k_{\max}$.

5.7 Principal Components Regression Optimization

The generalized cyclic coordinate method can also be applied to principal components regression (PCR). This type of regularization has three parameters: the soft threshold center, s_c, the threshold width, s_w, and the number of principal components, n_σ. Since there are only three regularization parameters and since the gradient can not be calculated for these parameters, they are optimized using the semi-global line search. The metric parameters are optimized using an estimate of the gradient obtained by dithering. In this section the cross-validation error is written as $\mathrm{CVE}(k, s_c, s_w, n_\sigma, \lambda)$. The complete algorithm is described below.

Algorithm 5.4: Principal Components Regression Optimization

1. Define s_c as the soft threshold center, s_w as the soft threshold width, n_σ as the number of principal components, λ as the metric parameters, and k as the number of neighbors used to build the model. Define $\Phi = \{\frac{1}{\phi_{\max}}, \ldots, \phi_{\max}\}$ as a finite set of real-valued amplification factors.
2. Initialize the stopping criteria:
   $n_i := 0$,
   $n_r := 0$.
3. Store the current value of the CVE:
   $\mathrm{CVE}_{\mathrm{prev}} := \mathrm{CVE}(k, s_c, s_w, n_\sigma, \lambda)$.
4. Optimize the number of neighbors by an exhaustive line search:
   $k := \underset{k_{\min} \le i \le k_{\max}}{\operatorname{argmin}} \mathrm{CVE}(i, s_c, s_w, n_\sigma, \lambda)$.
5. Optimize number of principal components by an exhaustive line search:
   $n_\sigma := \underset{0 \le i \le k}{\operatorname{argmin}} \mathrm{CVE}(k, s_c, s_w, i, \lambda)$.
6. Optimize soft threshold center by a semi-global line search,
   $\phi^\star := \underset{\phi \in \Phi}{\operatorname{argmin}} \mathrm{CVE}(k, \phi s_c, s_w, n_\sigma, \lambda)$,
   and update the parameter value,
   $s_c := \phi^\star s_c$.
7. Optimize soft threshold width by a semi-global line search,
   $\phi^\star := \underset{\phi \in \Phi}{\operatorname{argmin}} \mathrm{CVE}(k, s_c, \phi s_w, n_\sigma, \lambda)$,
   and update the parameter value,
   $s_w := \phi^\star s_w$.
8. Perform gradient-based optimization.
   For $i = 1$ to n_u,
   8.1 Estimate the gradient by dithering:
   $\nabla_\lambda \mathrm{CVE}(k, s_c, s_w, n_\sigma, \lambda)$.
   8.2 Use the gradient to calculate a new direction of descent, g.

8.3 Perform a line search:
$\alpha := \underset{\alpha \geq 0}{\operatorname{argmin}} \operatorname{CVE}(k, s_c, s_w, n_\sigma, \lambda + \alpha g)$.

8.4 Update the parameters:
$\lambda := \lambda + \alpha g$.

8.5 Next i.

9. Update count of iterations:
$n_i := n_i + 1$.

10. If $n_i = n_{i,\max}$, then the number of cycles has reached the user-specified limit. Exit function.

11. If $\operatorname{CVE}(k, s_c, s_w, n_\sigma, \lambda) \neq \operatorname{CVE}_{\text{prev}}$, then the optimization has not yet converged. Goto 3.

12. If $n_r = n_{r,\max}$, then the semi-global range has been reduced the user-specified number of times. Exit function.

13. Reduce the range for the semi-global line search,
$\phi_{\max} := 1 + (\phi_{\max} - 1)/2$,
recalculate the set of amplification factors,
$\Phi := \{\frac{1}{\phi_{\max}}, \ldots, \phi_{\max}\}$,
and increment the count of reductions,
$n_r := n_r + 1$.

14. Goto 3.

The new user-specified parameters for this algorithm are the initial values for the metric weights λ, the initial threshold center s_c, the initial threshold width s_w, the initial number of principal components n_σ, and the range of values for k, $k_{\min}$ and $k_{\max}$.

5.8 Discussion

The method of parameter optimization discussed in this chapter is not without its disadvantages. Although the iterative algorithms discussed in this chapter can only improve the initial parameters provided by the user, the presence of local minima may limit the degree of improvement.

Another drawback of parameter optimization is that the user must specify many more parameter values than if optimization had not been used. In addition to choosing the initial model parameter values, the user must also specify values for the optimization algorithm parameters. For experienced users this may be considered an advantage since they have more control of the optimized model, but more often users will feel overwhelmed as they try to pick parameter values with little intuition of how they will ultimately affect the model accuracy.

Fortunately, if an optimization algorithm is implemented correctly it can *only* improve the model accuracy as compared to the values that would have been chosen by the user. Although it may require the user to pick many parameter values, the model accuracy is much less sensitive to the optimization parameter values than the model parameter values. This enables even inexperienced users to construct more accurate local models than would be possible if optimization were not employed.

5.9 Summary

This chapter discussed two new optimization algorithms for vectored ridge regression (VRR) and principal components regression (PCR). Both algorithms can only improve model performance since they are guaranteed to not increase the cross-validation error.

There are two significant disadvantages these algorithms: they can require substantial preprocessing and they require the user to pick the algorithm parameter values in addition to the initial values for the model parameters. However, the algorithms were designed so that the algorithm parameters control the amount of preprocessing required for optimization. This shifts the burden on the user from picking good model parameter values to picking good algorithm parameter values that make the appropriate tradeoff between computational resources and model accuracy. For example, instead of specifying the number of neighbors, the user is required to specify a range of good values, $k_{\min}$ and $k_{\max}$. If the user picks a large range the preprocessing requirements will be increased but the model may be more accurate.

There are several improvements that can be made to this approach when local models are used for chaotic time series prediction. These are discussed in Chapter 6 along with two new algorithms for parameter optimization that are designed specifically for the time series prediction problem. Examples of these algorithms applied to a variety of data sets are given in Chapter 7.

Chapter 6

Chaotic Time Series Prediction

Stocks have reached what looks like a permanently high plateau.
—Professor of Economics, Yale University, 1929.

This chapter discusses how local models can be used for time series prediction. This includes several modifications to the optimization algorithms and the model structure that increase the predictive accuracy and reduce the computation required.

This chapter is organized as follows. Section 6.1 describes assumptions about the source of the time series. Section 6.2 introduces local modeling for time series prediction with an example. Section 6.3 describes the importance of Takens' theorem. Section 6.4 describes a new model parameter and metric designed specifically for the time series prediction problem. Section 6.5 describes how to improve the accuracy of multi-step predictions. Section 6.6 gives two new optimization algorithms for vectored ridge regression (VRR) and principal components regression (PCR) specifically designed for the time series prediction problem.

6.1 Process Model and Assumptions

Continuous-time chaotic time series are generated by nonlinear dynamic systems that can be described by a set of nonlinear ordinary differential equations,

$$\begin{aligned} \dot{z}_t &= f_c(z_t), \\ y_t &= g_c(z_t), \end{aligned} \tag{6.1}$$

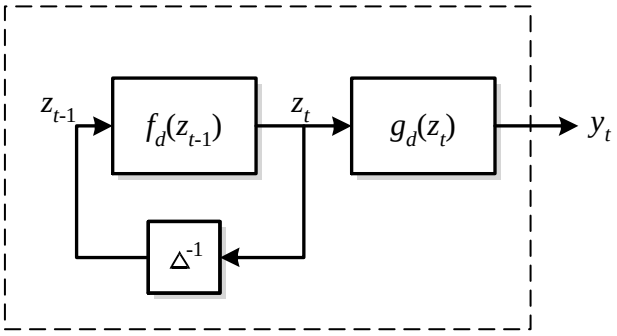


Figure 6.1: Process model for time series prediction. The operator $\triangle^{-1}$ represents a unit delay. The dashed box is used to emphasize that the equations have no external driving inputs and the system is essentially a nonlinear oscillator. Everything inside the dashed box is unknown to the user.

where $z_t \in \mathbb{R}^{n_s}$ is the state of the system, n_s is the order of the system, and y_t, a scalar, is the system output. The functions $f_d(\cdot)$ and $g_d(\cdot)$ must be smooth, in the sense that they have continuous and bounded partial derivatives, the system must be time invariant, and the system must have a finite number of equilibria. These conditions ensure that Takens' theorem applies (discussed in Section 6.3) and that the Lipschitz condition is satisfied, which implies the solution to Equation 6.1 exists and is unique [124, pp. 68–72].

An equivalent discrete-time system also exists,

$$\begin{aligned} z_t &= f_d(z_{t-1}), \\ y_t &= g_d(z_t), \end{aligned}$$

where

$$\begin{aligned} f_d(z_{t-1}) &\triangleq \int_{t-1}^{t} f_c(z_\tau)\, d\tau, \\ g_d(z_t) &\triangleq g_c(z_t). \end{aligned}$$

Without loss of generality, the sampling period is assumed to be one in this chapter for ease of presentation. Since the system has no external driving inputs, this system can be thought of as a nonlinear oscillator, as illustrated in Figure 6.1.

6.1.1 Chaotic Time Series

The system described in the previous section is only capable of a few types of behavior: convergent, divergent, periodic, quasi-periodic, or chaotic. Convergent time series converge

to a constant with time, $y_t \to c$ as $t \to \infty$, which makes them relatively easy to predict. Divergent time series increase in amplitude with time, $|y_t| \to \infty$ as $t \to \infty$, and can be very difficult to predict. Although there are some interesting applications that require short term prediction of these types of transient time series, the method described here is not well-suited to these types of problems.

Periodic time series have a fundamental period T such that $y_t = y_{t+T}$. Quasi-periodic time series are defined as a sum of two or more periodic time series such that the ratio of at least one of the pairs of fundamental periods is not a rational number[1]. The method described in this chapter can be successfully applied to these types of time series, but other, simpler methods work just as well.

Chaotic time series are similar to divergent time series because on average the initial separation of two similar states, z_0 and $z_0 + \delta$, increases exponentially with time, $||f_d^t(z_0) - f_d^t(z_0 + \delta)|| \propto \alpha^{-t}$ for small t, small δ, and some $\alpha > 1$. However, unlike divergent time series, the amplitude of chaotic time series is bounded. The rapid separation of close trajectories makes long term predictions of chaotic time series very difficult. Accurate short term predictions are possible, but the accuracy is very sensitive to the method of prediction.

6.1.2 Noise

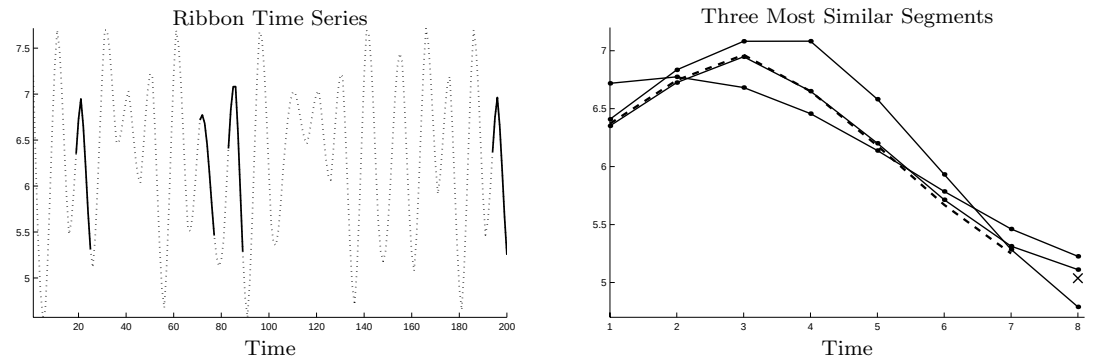
Many researchers have considered time series generated by nonlinear dynamic systems that contain noise,

$$\begin{aligned} \dot{z}_t &= f_c(z_t, u_t), \\ y_t &= g_c(z_t, v_t), \end{aligned}$$

where $u_t \in \mathbb{R}^{n_s}$ is called process noise and $v_t \in \mathbb{R}^1$ is called measurement or observation noise. Both types of noise are modeled as white noise processes with zero mean.

The majority of research on this topic attempts to remove the noise by finding a smoothed time series that satisfies the equations given in the previous section. The prediction methods described in this chapter may produce more accurate predictions if these noise reduction methods are applied to the time series as a preprocessing step, but the methods do not require this step. The interested reader is referred to the many papers on this topic for more information [30, 125–131].

[1] This guarantees that the series never repeats and hence the term *quasi*-periodic.



(*a*) (*b*)

Figure 6.2: (*a*) Chaotic time series with 200 points. The last darkened segment shows the points immediately preceding the point to be predicted. The other three darkened segments are the three closest segments in the time series. (*b*) The points immediately preceding the prediction point (dashed line) and the three closest segments (solid lines), all aligned in time. The $\times$ shows the prediction generated by averaging the points immediately after the three neighboring segments.

6.2 Example of Local Modeling for Prediction

Local models generate predictions by finding segments of the time series that closely resemble the segment immediately preceding the point to be predicted. The prediction is an estimate of the average change that occurred immediately after the neighboring segments. Figure 6.2 gives an illustration of this process applied to data recorded from a magnetoelastic ribbon[2].

6.3 Takens' Theorem

Takens' theorem states that there is a unique one-to-one mapping between the state of a dynamic system, z_t in Equation 6.1, and a finite window of the time series [132]. This window is called a time delay embedding,

$$x_t \triangleq [y_t, y_{t-\delta}, \ldots, y_{t-(n_d-1)\delta}]^{\mathrm{T}}, \tag{6.2}$$

where n_d is the embedding dimension and δ is the embedding delay. Takens' work was later generalized and shown to apply to a broader class of systems by Sauer *et al.* [133].

[2]This data used to be publicly available from Georgia Tech at *http://∼irobot.physics.gatech.edu*, but it was not available at the time of writing.

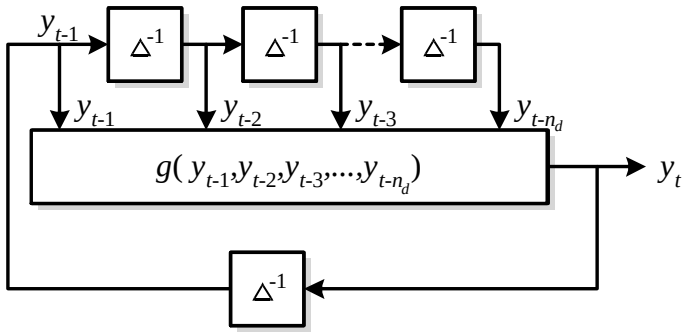


Figure 6.3: Equivalent process model for time series prediction. Note that the inputs to the function $g(\cdot)$ are known, only the function itself is unknown. Here the embedding delay was set equal to one, $\delta = 1$, for the sake of simplicity.

In order for the theorem to hold the embedding dimension must satisfy $n_d \geq 2n_s + 1$, where n_s is the order of the system.

Takens' theorem implies that if the assumptions described in Section 6.1 are satisfied, there exists a function $g(\cdot)$ such that

$$\begin{aligned} y_{t+1} &= g\left([y_t, y_{t-\delta}, \ldots, y_{t-(n_d-1)\delta}]^{\mathrm{T}}\right), \\ &= g(x_t). \end{aligned} \tag{6.3}$$

This is an important theorem for time series prediction because it implies perfect predictions are possible using only a finite segment of the values immediately preceding the point to be predicted. This idea is illustrated by Figure 6.3, which shows a diagram of a system that is equivalent to the system shown by Figure 6.1.

Given the process model described by Equation 6.1 and illustrated by Figure 6.1, even if the functions $f_d(z_t)$ and $g_d(z_t)$ were known, prediction would not be possible because the state of the system z_t is unknown to the user. The inputs to the equivalent system described by Equation 6.3 are available to the model and only the function $g(\cdot)$ is unknown.

The time series provides many examples of the mapping $y_{t+1} = g(x_t)$. If $\delta = 1$, these mappings can be written as

$$\begin{aligned} y_{n_d+1} &= g([\, y_{n_d} \quad, y_{n_d-1}, \ldots, y_1 \quad\;\;]^{\mathrm{T}}), \\ y_{n_d+2} &= g([\, y_{n_d+1}, y_{n_d} \quad, \ldots, y_2 \quad\;\;]^{\mathrm{T}}), \\ &\vdots \\ y_{n_y} &= g([\, y_{n_y-1}, y_{n_y-2}, \ldots, y_{n_y-n_d}]^{\mathrm{T}}). \end{aligned} \tag{6.4}$$

Thus, the chaotic time series prediction problem can be framed as a nonlinear modeling problem.

A more accurate model can be constructed if the knowledge that the data comes from a chaotic time series is embedded in the model. The following sections discuss some of the nuances of using nonlinear models for time series prediction and discuss techniques of how to incorporate this knowledge into local models to create more accurate predictions.

6.3.1 Intrinsic Manifold Dimension

One of the consequences of Takens' theorem is that all of the points in the reconstructed data set described by Equation 6.4 lie on a nonlinear manifold with a dimension no more than $2n_s + 1$ where n_s is the order of the system described by Equation 6.1. This is analogous to collinearity because each input vector contains at most $2n_s + 1$ independent variables regardless of the value of n_d, the size of the vector.

There are several significant consequences of this fact. First, if the order of the system is fairly small, say 3–5, all of the points will lie on a low-dimensional nonlinear manifold. As discussed in Chapter 3, this ensures that the fast nearest neighbor algorithms can work efficiently.

Second, this side-steps the curse of dimensionality because even if the dimension of the space is very large, the dimension of the surface that contains the data set remains small. Thus, it does not require an exorbitant number of points to densely populate the space.

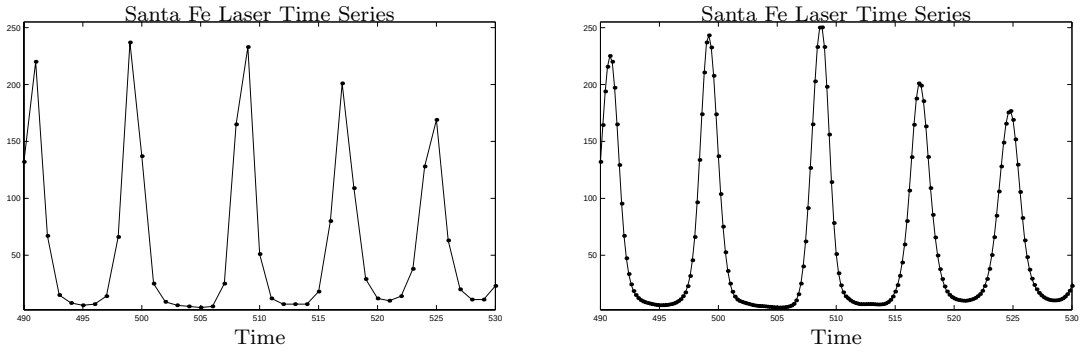
Third, this makes regularization especially important because, if the manifold is locally linear, all of the neighboring points will lie in a linear subspace and the matrix product $A^{\mathrm{T}}A$, which is inverted during the calculation of the linear coefficients, will be nearly singular.

6.3.2 Upsampling

One of the differences between the nonlinear modeling problem and the time series prediction problem is that the number of points in the data set can be increased by upsampling the time series. Upsampling is a standard one-dimensional interpolation technique and there are many good ways to do it.

For time series that are sparsely sampled, upsampling causes the neighbors to be closer to the query point and can significantly increase the model accuracy. The main disadvantage is the additional computational cost of finding the nearest neighbors in a larger data set.

If the time series is upsampled too much, the nearest neighbors may all be adjacent in time. One technique for solving this problem is to find the nearest trajectory segments,



(*a*) Original (*b*) Upsampled by five

Figure 6.4: This figure shows the Santa Fe Laser time series before (*a*) and after (*b*) upsampling by a factor of five.

instead of the nearest neighbors [9]. However, this technique is computationally intensive and similar accuracy can be achieved by increasing the size of the neighborhood.

Since upsampling increases accuracy at the cost of additional computation, the upsampling rate is best left as a user-specified parameter. A visual examination of the power spectrum and the raw time series can often help the user make a good choice.

6.3.3 Window Length

Takens' theorem assumes that the time series is noise-free; in practice this condition is rarely met and the selection of the embedding delay, δ, and the embedding dimension, n_d, may critically affect how accurately the time delay embedding reconstructs the state of the system. Many researchers have recognized this problem and proposed methods to find δ and n_d for finite time series with and without noise [27,134–136]. The goal of these methods is usually to find the values of these parameters that minimize the embedding dimension n_d without sacrificing the accuracy of the reconstruction. Although a compact reconstruction is efficient computationally, it does not necessarily maximize prediction accuracy; a point that is often overlooked.

There is some evidence that the accuracy of the reconstruction is not sensitive to the value of n_d for finite noise-free time series so long as n_d is sufficiently large. Kugiumtzis makes a strong case that the most important consideration in choosing the embedding parameters n_d and δ, is the window length, $\omega \triangleq n_d\,\delta$ [27]. As long as ω is sufficiently large and n_d is not too small, a wide range of values of δ and n_d will create an accurate reconstruction of the system state.

Before the model can be constructed, the embedding parameters must be chosen to construct the data set for the local model. A good value for the window length, ω, can often be chosen by visual inspection of the time series. Typically, noise-free univariate time series generated by the dynamic system in Equation (6.1) will contain periodic oscillations of roughly the same period; ω should be chosen to span several of these oscillations. The only disadvantages of choosing a larger value for ω than necessary are that it may increase the computation and it reduces the number of points that are available to build the model. If the time series contains n_y points, the number of points in the data set is given by

$$\begin{aligned} n_p &= n_y - \omega, \\ &= n_y - n_d\,\delta. \end{aligned}$$

For large time series $n_p \approx n_y$ and the value of ω is not critical as long as it is not too small. For smaller time series the choice is more important.

6.3.4 Choosing the Embedding Parameters

Once the window length ω has been selected, the embedding delay, δ, must be chosen. Although many researchers have proposed algorithms for minimizing the embedding dimension, there is a good reason to pick a small value for δ, which increases the embedding dimension. Consider the square of the Euclidean distance between two points,

$$\mathrm{D_E}\left(x_\tau, x_t\right)^2 = \sum_{i=0}^{n_d-1} \left(y_{\tau-i\delta} - y_{t-i\delta}\right)^2, \tag{6.5}$$

where x_τ is defined by Equation 6.2. In this case, the distance is an approximation of the integrated squared error,

$$\begin{aligned} \mathrm{ISE}(x_\tau, x_t) &\triangleq \int_0^{n_d\delta} \left(x_{\tau-\upsilon} - x_{t-\upsilon}\right)^2 d\upsilon, \\ &= \lim_{\Delta\upsilon\to 0} \sum_{i=0}^{\frac{n_d\delta}{\Delta\upsilon}-1} \left(x_{\tau-i\,\Delta\upsilon} - x_{t-i\,\Delta\upsilon}\right)^2 \Delta\upsilon, \\ &= \lim_{\delta\to 0} \sum_{i=0}^{n_d-1} \left(x_{\tau-i\delta} - x_{t-i\delta}\right)^2 \delta, \\ &= \lim_{\delta\to 0} \mathrm{D_E}\left(x_\tau, x_t\right)^2 \delta, \\ &\approx \mathrm{D_E}\left(x_\tau, x_t\right)^2 \delta, \end{aligned}$$

and finally,

$$\mathrm{D_E}\left(x_\tau, x_t\right)^2 \approx \frac{1}{\delta}\,\mathrm{ISE}(x_\tau, x_t). \tag{6.6}$$

For a fixed window length, the choice of δ governs the accuracy of the estimated $\mathrm{ISE}(\cdot)$. Small values of δ will increase the accuracy of the estimated $\mathrm{ISE}(\cdot)$, which will usually increase the local model accuracy. However, small values of δ will also increase the computation required to calculate $\mathrm{D_E}\left(x_\tau, x_t\right)$. In light of this tradeoff, the user should pick a value for δ that is small, but within the limits of the computational budget. The same arguments and tradeoffs apply when the diagonally weighted Euclidean metric is used.

6.4 Local Model Parameterization

This section describes a new model parameter for time series prediction than can improve model accuracy and reduce computation. A new exponential metric is also described to eliminate the need for gradient estimation in principal components regression.

6.4.1 Number of Local Inputs

For the time series prediction problem the model input variables have a natural order that can be exploited to build more accurate models. Since the input variables are arranged as

$$x_t = \left[y_t, y_{t-\delta}, \ldots, y_{t-(n_d-1)\delta}\right]^{\mathrm{T}} \tag{6.7}$$

the latter elements of the input vector will have less influence on the process output, y_{t+1}. This is especially true of chaotic time series since neighboring states diverge from one another at an exponential rate, on average. In this case, it may be better to build the local linear model without the latter input variables, which decreases the variance of the local model output.

The cyclic optimization algorithm in the previous chapter can easily accommodate this modification by treating the number of local inputs, n_ℓ, as a new model parameter to be optimized. This approach has the additional benefit of letting the optimization algorithm decide between a local averaging model, $n_\ell = 0$, and a local linear model, $n_\ell > 0$.

6.4.2 Exponential Metrics

Choosing an appropriate distance metric is an important decision that can strongly affect the accuracy of local models for time series prediction [26,120,121]. The most popular choice is the Euclidean distance described by Equation 6.5, but some researchers have suggested alternatives.

Murray proposed using a exponentially weighted diagonal metric[3],

$$\mathrm{D_M}(x_\tau, x_t)^2 \triangleq \sum_{i=0}^{n_d-1} \beta^i \left(y_{\tau-i\delta} - y_{t-i\delta}\right)^2,$$

where β is a scalar between zero and one [120].

This metric is intuitively appealing because the components of x_t closest in time to the prediction are given exponentially more weight than the distant points. It is especially appropriate for chaotic systems where neighboring states are known to diverge exponentially with time. Since the exponential metric is equal to the standard Euclidean metric when $\beta = 1$, an appropriate choice of β can only improve the model accuracy as compared to the Euclidean metric.

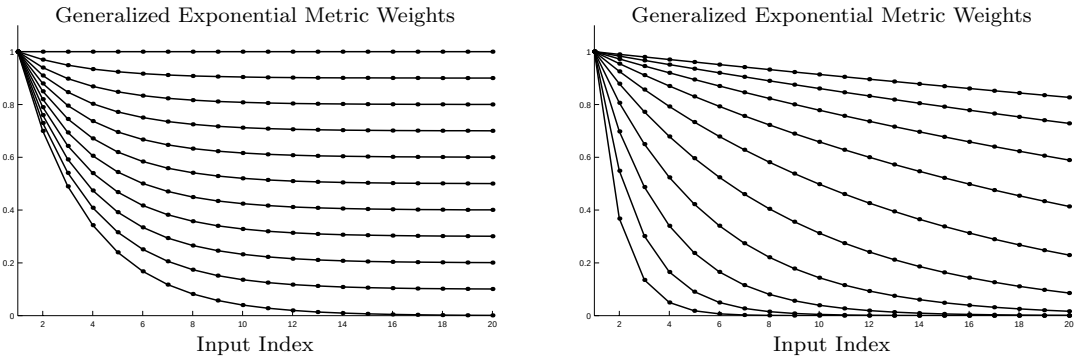
This metric is less general than the diagonally weighted Euclidean metric described in the previous chapter, but it also has fewer parameters and can be optimized more quickly. This is an especially attractive option for principal components regression (PCR) since the single-parameter optimization does not require gradient estimation, which is computationally intensive.

A more general form of the exponential metric is

$$\mathrm{D_G}(x_\tau, x_t)^2 \triangleq \sum_{i=0}^{n_m} \Big((1-\rho) * e^{-\beta * i} + \rho\Big) \left(y_{\tau-i\delta} - y_{t-i\delta}\right)^2,$$

where $n_m < n_d$ is the number of terms in the summation, β is a scalar that controls the rate of decay, and ρ controls the amount of decay. As illustrated by Figure 6.5, this metric has considerably more flexibility than the exponential metric but still has few enough parameters to be efficiently optimized by the cyclic coordinate method. An algorithm for optimizing these parameters for principal components regression is described in Section 6.6.2.

[3]This metric is in a different form than that proposed by Murray, but it is mathematically equivalent.



(*a*) $\beta = 0.7$ (*b*) $\rho = 0$

Figure 6.5: (*a*) The metric weights $((1-\rho) * \beta^i + \rho)$ for $\beta = 0.7$ and various values of ρ from $\rho = 0$ for the bottom line to $\rho = 1$ for the top line. (*b*) The same metric weights for $\rho = 0$ and various values of β from $\beta = 0.1$ for the bottom line to $\beta = 1$ for the top line.

Previous Work

Although the choice of the metric can drastically affect the accuracy of local models, very little has been published on this topic in the field of chaotic time series prediction. The only publications known to the author are briefly summarized in this section.

Farmer and Sidorowich suggested a fixed $\lambda_i = e^{-hi}$, where h is the metric entropy, which they report is in some sense linearly optimal [34]. In addition to the exponential metric, Murray also investigated a two-parameter metric that used a tri-diagonal weighting matrix [120].

Kugiumtzis compared L_1, L_2 and L_∞ norms, mostly for the purpose of estimating correlation dimension [137]. Kugiumtzis also performed a brief analysis using local linear models for prediction and reported that there are not significant differences in the prediction errors for each of the three norms. Casdagli and Weigend reported that model accuracy is very sensitive to the choice of the embedding dimension when the Euclidean metric is used [48].

Tanaka *et al.* developed an elegant approximation to an optimal metric with a rank-1 weighting matrix [121]. This metric has no user-specified parameters, is computationally inexpensive, and initial results indicate it may work better than the Euclidean metric or Murray's metric. The same metric was also developed independently by Garcia *et al.* [26].

Fraedrich and Ruckert recently proposed a method for adapting the diagonally weighted Euclidean metric [122]. Unlike the gradient-based optimization approach taken in the

previous chapter, this version applies only to the nearest neighbor model[4] and uses an *ad hoc* optimization algorithm.

6.5 Multi-Step Predictions

This section discusses two different approaches to generating predictions for more than one step ahead and the importance of picking an appropriate error measure for iterative models.

6.5.1 Iterated versus Direct Prediction

Suppose we are given a time series, $[y_1, y_2, \ldots, y_{n_y}]$, and asked to predict n_a steps ahead. *Direct prediction* is the method in which a model is built to directly predict y_{t+n_a},

$$\begin{aligned} \hat{y}_{t+n_a} &= \hat{g}_\gamma(x_t), \\ &= \hat{g}_\gamma([y_t, y_{t-\delta}, \ldots, y_{t-(n_d-1)\delta}]^{\mathrm{T}}), \end{aligned}$$

where $\hat{g}_\gamma(\cdot)$ is the model defined by the parameters stored in the vector γ.

For *iterated prediction*, a model is built to predict one step ahead and the prediction is then used as part of the input vector to predict two steps ahead and so on. In each step, the model estimates y_{t+1} from x_t and this is then used to estimate x_{t+1}. For example, if $\delta = 1$,

$$\begin{aligned} x_t^{\mathrm{T}} &= \left[y_t, y_{t-1}, y_{t-2}, \ldots, y_{t-(n_d-1)}\right], \\ \hat{y}_{t+1} &= \hat{g}_\gamma(x_t), \\ \hat{x}_{t+1}^{\mathrm{T}} &= \left[\hat{y}_{t+1}, y_t, y_{t-1}, \ldots, y_{t-(n_d-2)}\right], \\ \hat{y}_{t+2} &= \hat{g}_\gamma(\hat{x}_{t+1}), \\ \hat{x}_{t+2}^{\mathrm{T}} &= \left[\hat{y}_{t+2}, \hat{y}_{t+1}, y_t, \ldots, y_{t-(n_d-3)}\right], \\ \hat{y}_{t+3} &= \hat{g}_\gamma(\hat{x}_{t+2}), \\ &\vdots \end{aligned} \tag{6.8}$$

This process is iterated for n_a steps finally producing the prediction $\hat{y}_{t+n_a}$.

There has been much debate over which method is better. Direct prediction is questionable because a function that maps n_a steps into the future will usually be more complicated

[4]The nearest neighbor model uses the target value of the nearest neighbor as the model output. This is equivalent to local averaging with $k = 1$.

and thereby harder to model than one that predicts a single step into the future [35]. Iterated prediction is questionable because it does not take into account the accumulated errors in the input vector, $\hat{g}_\gamma(x_{t+c}) \neq \hat{g}_\gamma(\hat{x}_{t+c})$.

Most researchers have found that iterated prediction is more accurate than direct prediction [18, 34, 35, 136]. As a compromise, Sauer suggested averaging the direct and iterated predictions [37]. In the next section, an improvement to iterated prediction is described.

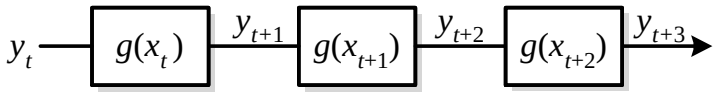
6.5.2 Multi-Step Error Estimation

In framing the time-series prediction problem as a nonlinear modeling problem, the natural error measure for local models is the cross-validation error,

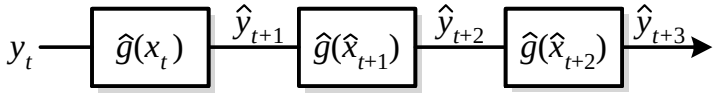
$$\text{CVE}(\gamma) \triangleq \sum_{i=1}^{n_c} p\left(y_{c(i)} - \hat{g}_\gamma^-(x_{c(i)})\right),$$

where n_c controls the accuracy of the estimated error, $c(i)$ is the data set index of the ith cross-validation point, $p(\cdot)$ is the user-specified penalty function, and the vector γ contains all of the parameters that define the model. The output, $\hat{g}_\gamma^-(x_{c(i)})$, is from a local model constructed with the $c(i)$th point omitted from the data set, but used as the model input.

Because this minimizes the error of predicting one-step ahead, it is called the one-step cross-validation error (OSCVE). A crucial disadvantage of using one-step cross-validation error is that it does not take into account the effect of the errors in the input vector that



(a) Takens Equivalent



(b) Multiple Iterative Models

Figure 6.6: (a) The Takens' equivalent of the system that generated the time series. (b) The errors in the iterative model are caused by modeling errors, $\hat{g}(\cdot) \neq g(\cdot)$, and by the errors in the input vector from previous predictions, $\hat{x}_{t+i} \neq x_{t+i}$ for $i > 1$.

occur with iterated prediction; the parameter values that minimize the OSCVE are generally not the same values that maximize the model performance for predicting n_a steps ahead. The diagram in Figure 6.6 illustrates this idea.

The problem of neglecting the errors in the model input vector is especially acute for local linear models. If some of the input variables are collinear, the ill-conditioned matrix inverse of $A^{\mathrm{T}}A$ will cause the magnitude of the linear model coefficients to be very large. If the model input vector, or query, happens to lie in the column space of A, the output may be accurate despite the large coefficients. However, if the model input vector is only an approximation, the errors in the approximation, $\hat{x}_t - x_t$, will be greatly amplified by the model coefficients.

A more appropriate choice for optimization is to use an error measure that reflects the true cost for making iterative predictions where the *true cost* depends on the application. In many cases an average (possibly weighted) model accuracy over n_a-steps ahead is appropriate,

$$\overline{\mathrm{CVE}}_{n_a}(\gamma) \triangleq \frac{1}{n_c n_a} \sum_{i=1}^{n_c} \sum_{j=1}^{n_a} p\left(y_{c(i)+j} - \hat{g}_\gamma^-\left(x_{c(i)+j-1}\right)\right),$$

where n_a is the number of steps ahead over which the average error is measured. This error measure is called multi-step cross validation error (MSCVE).

The main disadvantage of MSCVE is the requirement for additional computation. If the amount of computation is restricted, the user must tradeo ff accuracy of the estimated $\overline{\mathrm{CVE}}$, determined by n_c, for the number of steps ahead over which the estimate is taken, determined by n_a.

6.6 Model Optimization

The previous chapter described two optimization algorithms for vectored ridge regression (VRR) and principal components regression (PCR). This section describes two new versions of each of these algorithms that include the innovations for time series prediction described in this chapter.

6.6.1 Vectored Ridge Regression Optimization

The VRR optimization algorithm described in the previous chapter can be tailored to time series prediction problems by making two modifications: the number of local inputs is added

as an new model parameter to be optimized and the multi-step cross-validation error, $\overline{\text{CVE}}$, is used as the measure of model accuracy.

The Error Gradient

The gradient of $\overline{\text{CVE}}$ is more difficult to derive and compute than for the nonlinear modeling problem because both the model $\hat{g}_\gamma(\cdot)$ and the model input vector[5]

$$\begin{aligned}\hat{x}_t &= \left[\hat{y}_t, \hat{y}_{t-1}, \ldots, y_{t-(n_d-1)}\right]^{\mathrm{T}}, \\ &= \left[\hat{g}_\gamma(\hat{x}_{t-1}), \hat{g}_\gamma(\hat{x}_{t-2}), \ldots, y_{t-(n_d-1)}\right]^{\mathrm{T}},\end{aligned}$$

depend on the model parameters[6] γ.

To simplify the discussion, the vector γ is defined in this section as the concatenation of the metric parameters, λ, and the ridge parameters, r, so that $\gamma^{\mathrm{T}} \triangleq [\lambda^{\mathrm{T}} \;\; r^{\mathrm{T}}]$; note that the model is also defined by k, the number of neighbors, and n_ℓ, the number of local inputs.

The gradient of $\overline{\text{CVE}}(k, n_\ell, \gamma)$ is given by

$$\nabla_\gamma \overline{\text{CVE}}(k, n_\ell, \gamma) = \frac{1}{n_c n_a} \sum_{i=1}^{n_c} \sum_{j=1}^{n_a} \nabla_\gamma p\left(y_{c(i)+j} - \hat{g}_\gamma^-(\hat{x}_{c(i)+j-1})\right).$$

If we define a new variable, $\varepsilon \triangleq y_{c(i)+j} - \hat{g}_\gamma^-(x_{c(i)+j-1})$, the gradient can be written as

$$\nabla_\gamma \overline{\text{CVE}}(k, n_\ell, \gamma) = \frac{1}{n_c n_a} \sum_{i=1}^{n_c} \sum_{j=1}^{n_a} -\frac{dp(\varepsilon)}{d\varepsilon} \nabla_\gamma \hat{g}_\gamma^-(\hat{x}_{c(i)+j-1}).$$

The lth element of the gradient of the model output, $\nabla_\gamma \hat{g}_\gamma^-(\hat{x}_{c(i)+j-1})$, is given by

$$\frac{\partial \hat{g}_\gamma^-(\hat{x}_{c(i)+j-1})}{\partial \gamma_l} = \underbrace{\left.\frac{\partial \hat{g}_\gamma^-(u)}{\partial \gamma_l}\right|_{u=\hat{x}_{c(i)+j-1}}}_{①} + \sum_{m=1}^{\min(n_d, j-1)} \underbrace{\frac{\partial \hat{g}_\gamma^-(\hat{x}_{c(i)+j-1})}{\partial \hat{x}_{c(i)+j-1,m}}}_{②} \underbrace{\frac{\partial \hat{x}_{c(i)+j-1,m}}{\partial \gamma_l}}_{③}. \tag{6.9}$$

The first term, ①, accounts for the effect of γ_l on the model estimate $\hat{g}_\gamma(u)$ for a constant input, u. This is derived in Appendix A. The first term in the summation, ②, is the partial derivative of the model output with respect to the model inputs and is also derived in Appendix A.

[5] In this section, the embedding delay is assumed to be one.

[6] Recall from Equation 6.8 that the elements of x_t are estimated only when the true value is not known.

Since the estimated input vector is created from previous model outputs,

$$\hat{x}_{c(i)+j-1,m} = \hat{g}_\gamma(\hat{x}_{c(i)+j-1-m}),$$

the last term in the summation of Equation 6.9, ③, is the lth element of the gradient from the previous step. Thus, the gradient can be calculated recursively,

$$\begin{aligned}
\frac{\partial \hat{g}_\gamma^-(x_{c(i)})}{\partial \gamma_l} &= \left.\frac{\partial \hat{g}_\gamma^-(u)}{\partial \gamma_l}\right|_{u=x_{c(i)}}, \\
\frac{\partial \hat{g}_\gamma^-(\hat{x}_{c(i)+1})}{\partial \gamma_l} &= \left.\frac{\partial \hat{g}_\gamma^-(u)}{\partial \gamma_l}\right|_{u=\hat{x}_{c(i)+1}} + \frac{\partial \hat{g}_\gamma^-(\hat{x}_{c(i)+1})}{\partial \hat{x}_{c(i)+1,1}} \frac{\partial \hat{g}_\gamma^-(x_{c(i)})}{\partial \gamma_l}, \\
\frac{\partial \hat{g}_\gamma^-(\hat{x}_{c(i)+2})}{\partial \gamma_l} &= \left.\frac{\partial \hat{g}_\gamma^-(u)}{\partial \gamma_l}\right|_{u=\hat{x}_{c(i)+2}} + \frac{\partial \hat{g}_\gamma^-(\hat{x}_{c(i)+2})}{\partial \hat{x}_{c(i)+2,1}} \frac{\partial \hat{g}_\gamma(\hat{x}_{c(i)+1})}{\partial \gamma_l} + \frac{\partial \hat{g}_\gamma^-(\hat{x}_{c(i)+2})}{\partial \hat{x}_{c(i)+2,2}} \frac{\partial \hat{g}_\gamma^-(x_{c(i)})}{\partial \gamma_l}, \\
&\vdots
\end{aligned}$$

This recursive approach enables the gradient to be calculated efficiently[7].

Algorithm

The time series version of the VRR optimization algorithm is given below. Except for the additional step to optimize n_ℓ and the use of MSCVE, this is identical to the algorithm described in Chapter 5. Here the multi-step cross-validation error is denoted as $\overline{\mathrm{CVE}}(k, n_\ell, \gamma)$ where k is the number of neighbors, n_ℓ is the number of local model inputs, and γ is a vector that contains the metric and ridge parameters.

Since the range for the variable n_ℓ is naturally bounded by $0 \leq n_\ell \leq n_d$, the only additional user-specified parameters are an initial value for n_ℓ and the number of steps included in $\overline{\mathrm{CVE}}$, n_a.

Algorithm 6.1: Time Series Vectored Ridge Regression Optimization

1. Define $\gamma^{\mathrm{T}} \triangleq [\lambda^{\mathrm{T}} \; r^{\mathrm{T}}]$ where λ are the metric parameters and r are the ridge parameters. Define k as the number of neighbors and n_ℓ as the number of local model inputs.
2. Initialize the stopping criteria:
   $n_i := 0$,
   $\overline{\mathrm{CVE}}_{\mathrm{prev}} := \infty$.

[7] Researchers have employed a similar approach for recurrent neural networks called real time recurrent learning [138]. Backpropagation through time, a closely related, and mathematically equivalent technique, has also been used [139].

3. Optimize the number of neighbors by an exhaustive line search:
   $k := \underset{k_{\min} \le i \le k_{\max}}{\operatorname{argmin}} \overline{\mathrm{CVE}}(i, n_\ell, \gamma)$.
4. Optimize the number of local inputs by an exhaustive search:
   $n_\ell = \underset{0 \le i \le n_d}{\operatorname{argmin}} \overline{\mathrm{CVE}}(k, i, \gamma)$.
5. For $i = 1$ to n_u,
   5.1 Calculate the gradient:
   $\nabla_\gamma \overline{\mathrm{CVE}}(k, n_\ell, \gamma)$.
   5.2 Calculate a new direction of descent, g.
   5.3 Perform a line search:
   $\alpha := \underset{\alpha \ge 0}{\operatorname{argmin}} \overline{\mathrm{CVE}}(k, n_\ell, \gamma + \alpha g)$.
   5.4 Update the parameters:
   $\gamma := \gamma + \alpha g$.
   5.5 Next i.
6. If $n_i = n_{i,\max}$, then the number of cycles has reached the user-specified limit. Exit function.
7. If $\overline{\mathrm{CVE}}(k, n_\ell, \gamma) = \overline{\mathrm{CVE}}_{\mathrm{prev}}$, then the optimization has converged. Exit function.
8. Update the stopping criteria:
   $n_i := n_i + 1$,
   $\overline{\mathrm{CVE}}_{\mathrm{prev}} := \overline{\mathrm{CVE}}(k, n_\ell, \gamma)$.
9. Goto 3.

6.6.2 Principal Components Regression Optimization

Like vectored ridge regression, the optimization algorithm for principal components regression (PCR) can be tailored to the task of chaotic time series prediction by including the number of local model inputs as an optimization parameter. The computationally intensive estimation of the gradient can also be eliminated by using the cyclic coordinate method to optimize the generalized exponential metric described in Section 6.4.2.

In addition to the user-specified parameters described in Chapter 5, this algorithm also requires the user to specify the initial values for the new parameters n_ℓ, n_m, $n_{m,\min}$, β, and ρ.

Here the multi-step cross-validation error is written as $\overline{\mathrm{CVE}}(k, n_\ell, s_c, s_w, n_m, \beta, \rho)$ to explicitly show the dependence on each of the model parameters. The complete algorithm is given below.

Algorithm 6.2: Time Series Principal Components Regression Optimization

1. Define s_c as the soft threshold center, s_w as the soft threshold width, n_σ as the number of principal components, λ as the metric parameters, k as the number of neighbors used to build the model, and n_ℓ as the number of local model inputs. Define $\Phi = \{\frac{1}{\phi_{\max}}, \ldots, \phi_{\max}\}$ as a finite set of real-valued amplification factors.

2. Initialize stopping criteria:
   $n_c := 0$,
   $n_r := 0$.

3. Update stopping criteria:
   $n_c := n_c + 1$,
   $\overline{\text{CVE}}_{\text{prev}} := \overline{\text{CVE}}(k, n_\ell, s_c, s_w, n_\sigma, n_m, \beta, \rho)$.

4. Optimize the number of neighbors by an exhaustive search:
   $k := \underset{k_{\min} \le i \le k_{\max}}{\text{argmin}} \ \overline{\text{CVE}}(i, n_\ell, s_c, s_w, n_\sigma, n_m, \beta, \rho)$.

5. Optimize the number of local model inputs by an exhaustive search:
   $n_\ell := \underset{0 \le i \le n_d}{\text{argmin}} \ \overline{\text{CVE}}(k, i, s_c, s_w, n_\sigma, n_m, \beta, \rho)$.

6. Optimize number of principal components by an exhaustive search,
   $n_\sigma := \underset{0 \le i \le k}{\text{argmin}} \ \overline{\text{CVE}}(k, n_\ell, s_c, s_w, i, n_m, \beta, \rho)$.

7. Optimize soft threshold center by a semi-global line search,
   $\phi^\star := \underset{\phi \in \Phi}{\text{argmin}} \ \overline{\text{CVE}}(k, n_\ell, \phi s_c, s_w, n_\sigma, n_m, \beta, \rho)$,
   and update the parameter value,
   $s_c := \phi^\star s_c$.

8. Optimize soft threshold width by a semi-global line search,
   $\phi^\star := \underset{\phi \in \Phi}{\text{argmin}} \ \overline{\text{CVE}}(k, n_\ell, s_c, \phi s_w, n_\sigma, n_m, \beta, \rho)$,
   and update the parameter value,
   $s_w := \phi^\star s_w$.

9. Optimize the metric exponent factor by a semi-global line search,
   $\phi^\star := \underset{\phi \in \Phi}{\text{argmin}} \ \overline{\text{CVE}}(k, n_\ell, s_c, s_w, n_\sigma, n_m, \phi\beta, \rho)$,
   and update the parameter value,
   $\beta := \phi^\star \beta$.

10. Optimize the metric offset by a semi-global line search,
    $\phi^\star := \underset{\phi \in \Phi}{\text{argmin}} \ \overline{\text{CVE}}(k, n_\ell, s_c, s_w, n_\sigma, n_m, \beta, \phi\rho)$,
    and update the parameter value,
    $\rho := \phi^\star \rho$.

11. Optimize the number of metric inputs by an exhaustive line search:
    $n_m := \underset{n_{m,\min} \le i \le n_d}{\text{argmin}} \ \overline{\text{CVE}}(k, n_\ell, s_c, s_w, n_\sigma, i, \beta, \rho)$.

12. If $n_c = n_{c,\max}$, then the number of cycles has reached the user-specified limit. Exit function.

13. If $\overline{\text{CVE}}(k, n_\ell, s_c, s_w, n_\sigma, n_m, \beta, \rho) \neq \overline{\text{CVE}}_{\text{prev}}$, then the optimization has not yet converged. Goto 2.

14. If $n_r = n_{r,\max}$, then the semi-global range has been reduced the user-specified number of times. Exit function.

15. Reduce the range for the semi-global line search,
$\phi_{\max} := 1 + (\phi_{\max} - 1)/2$,
recalculate the set of amplification factors,
$\Phi := \{\frac{1}{\phi_{\max}}, \ldots, \phi_{\max}\}$,
and increment the count of reductions,
$n_r := n_r + 1$.

16. Goto 3.

6.7 Summary

This chapter described several techniques for modifying the local model parameterization and optimization for chaotic time series prediction. These modifications included a new parameter that controls how many inputs are used to build the local model and a new three-parameter exponential metric specifically designed for chaotic time series. This metric also enables efficient optimization of principal components regression (PCR). The optimization algorithms were also modified to minimize the multi-step cross-validation error (MSCVE) to ensure accurate multi-step predictions.

The next chapter demonstrates these optimization algorithms on several chaotic time series and discusses some of the tradeoffs and issues that arise in practice.

Chapter 7

Case Studies

It is better to read the weather forecast before we pray for rain.
—Mark Twain

This chapter demonstrates the effectiveness of the algorithms described in the previous chapter on several time series prediction problems.

This chapter is organized as follows. Section 7.1 describes how the model accuracy was measured and lists the parameter values used for the examples in this chapter. Section 7.2 illustrates the validity of exponential metrics and Section 7.3 discusses the problem of local minima. Section 7.4 gives an example of the improvement in model accuracy due to using multi-step error estimation during model optimization. Section 7.5 illustrates the new methods of prediction on five chaotic time series taken from a variety of sources.

7.1 Introduction

It is difficult to compare the new methods to previous techniques for local modeling because previous techniques have relied on the expertise of the researcher to pick appropriate parameter values. The new methods use the parameter values selected by the user as a starting point and can only improve performance, as measured by the cross-validation error.

Instead this chapter focuses on the comparing the two methods of regularization, vectored ridge regression (VRR) and principal components regression (PCR), and the effect of minimizing the multi-step cross-validation error versus the one-step cross-validation error.

7.1.1 Normalized Mean Squared Error

The models in this chapter were constructed using only an initial segment of the time series. The remainder of the time series was used to estimate the model performance as a function of the number of steps predicted ahead.

The model accuracy was measured by the normalized mean squared error,

$$\mathrm{NMSE}(\tau) \triangleq \frac{1}{\sigma_y^2}\frac{1}{n_c}\sum_{i=1}^{n_c}\left(y_{v(i)+\tau} - \hat{g}_\gamma(\hat{x}_{v(i)+\tau-1})\right)^2, \tag{7.1}$$

where σ_y^2 is the sample variance of the time series, $y_{v(i)+\tau}$ is the $(v(i)+\tau)$th point in the time-series, $v(i)$ is the index of the first point in the ith validation segment, and $\hat{g}_\gamma(\hat{x}_{v(i)+\tau-1})$ is the prediction of the local model.

Similarly, the root normalized mean squared error is defined as

$$\mathrm{RNMSE}(\tau) \triangleq \sqrt{\mathrm{NMSE}(\tau)}.$$

$\mathrm{RNMSE}(\tau)$ is a convenient measure of error because it is independent of the scale of the time series and because it has a meaningful interpretation: $\mathrm{RNMSE}(\tau) = 1$ is the point at which the model prediction error is no better than predicting the sample mean $\bar{y}$, on average.

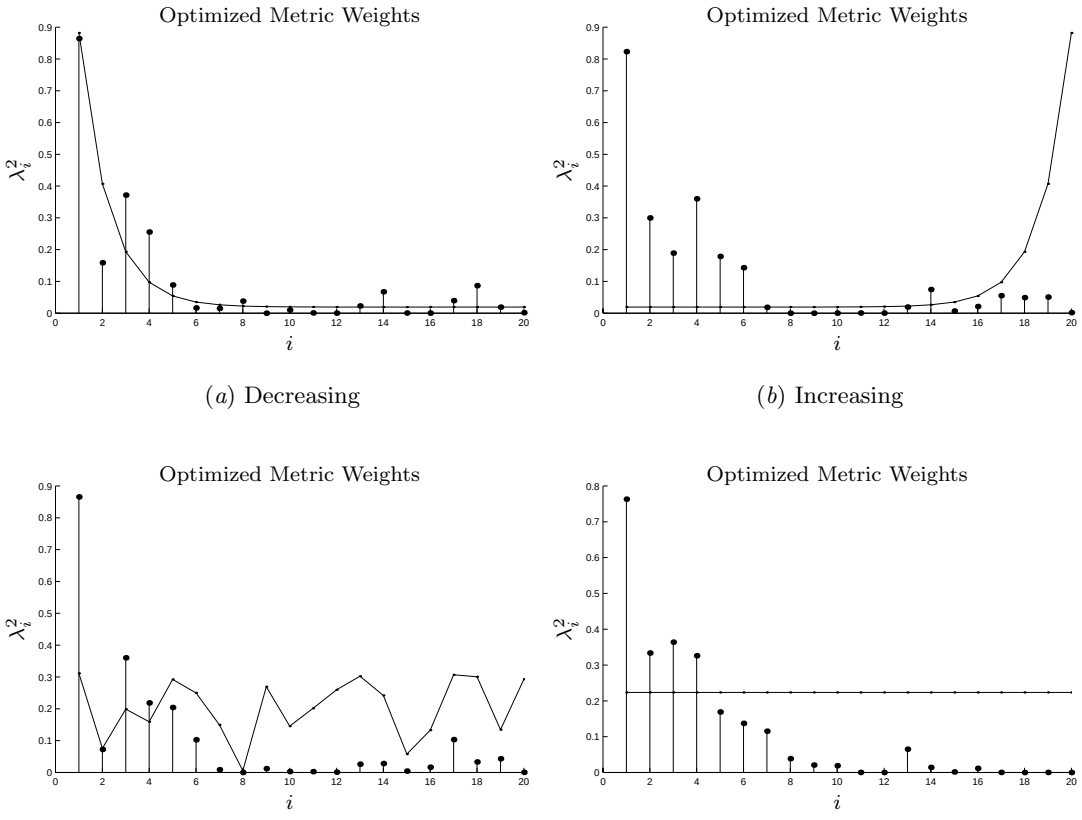
7.1.2 Parameter Values

Parameter	Value	Description
n_ℓ	15	Initial number of local inputs.
n_σ	5	Initial number of singular values.
s_c	0.0001	Initial center of PCR threshold.
s_w	0.05	Initial width of PCR threshold.
β	0.1	Initial metric exponent.
ρ	0.01	Initial metric offset.
n_m	20	Number of metric inputs.
n_c	250	Number of cross-validation points used for parameter optimization.
$k_{\min}$	5	Minimum number of neighbors.
$k_{\max}$	15	Maximum number of neighbors.
$\phi_{\max}$	10	Maximum value of the amplification factors.
n_d	20	Embedding dimension.
$n_{m,\min}$	10	Minimum exponential metric width.

Table 7.1: User-specified values chosen for the results reported here.

Unless otherwise noted, the user-specified parameter values used to generate the results in this chapter are listed in Table 7.1. For vectored ridge regression, the initial values for the metric weights and the ridge coefficients are shown in Figures 7.2*a* and 7.3*a*, respectively.

7.2 Exponential Metrics



(*a*) Decreasing (*b*) Increasing

(*c*) Random (*d*) Euclidean

Figure 7.1: Local averaging metric weights after optimization for four different initial weight configurations. In each figure the initial weights are shown by the solid line and the weights after optimization are shown by the stem plot.

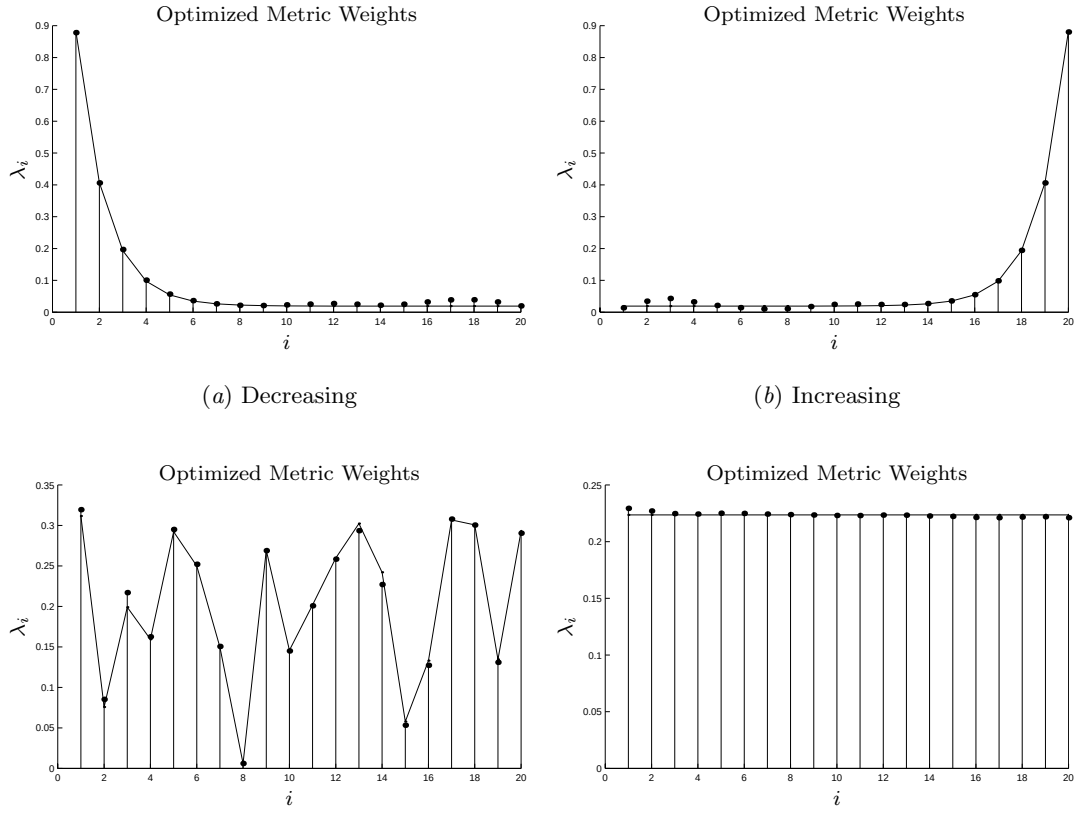
In the previous chapter it was suggested that exponential metrics are appropriate for time series prediction because the input vectors are composed of past values of the time series, $x_t = [y_t, y_{t-1}, \ldots, y_{t-n_p+1}]$. Intuitively, the initial elements should have a stronger relationship to y_{t+1} than the later elements.

To examine the validity of this hypothesis, a local averaging model was optimized to predict one step ahead by the VRR time series optimization algorithm using the first 3,000

points in the Lorenz time series. This process was repeated for four different initial values. The metric parameters before and after optimization are shown in Figure 7.1.

In this case the optimized weights all converged to a shape that was similar to a decreasing exponential, regardless of the values chosen for the initial weights. This gives empirical support to earlier work suggesting that exponential metrics are appropriate for time series prediction and suggests that the generalized exponential metric is a good choice for principal components regression.

7.3 Local Minima



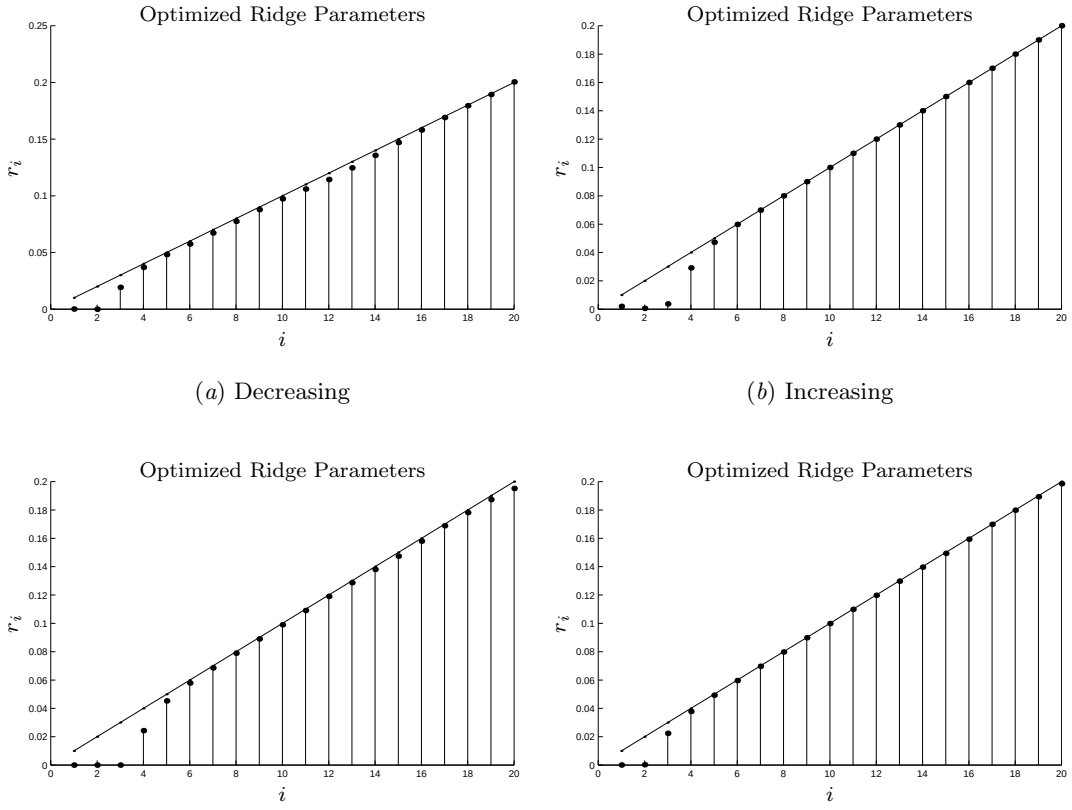
(*a*) Decreasing (*b*) Increasing

(*c*) Random (*d*) Euclidean

Figure 7.2: Ridge regression metric weights after optimization for four different initial weight configurations. In each figure the initial weights are shown by the line and the weights after optimization are shown by the stem plot.

Ridge regression is much more sensitive to the choice of initial metric weights than local averaging. Figure 7.2 shows the metric weights before and after optimization for the same initial weights used in the previous section. In this case, the optimization algorithm

became stuck in local minima and was unable to change the weights significantly. This illustrates the importance of choosing the initial weights appropriately for ridge regression. For all of the results reported here the weights were initialized by the decreasing exponential shown in Figure 7.2*a*.



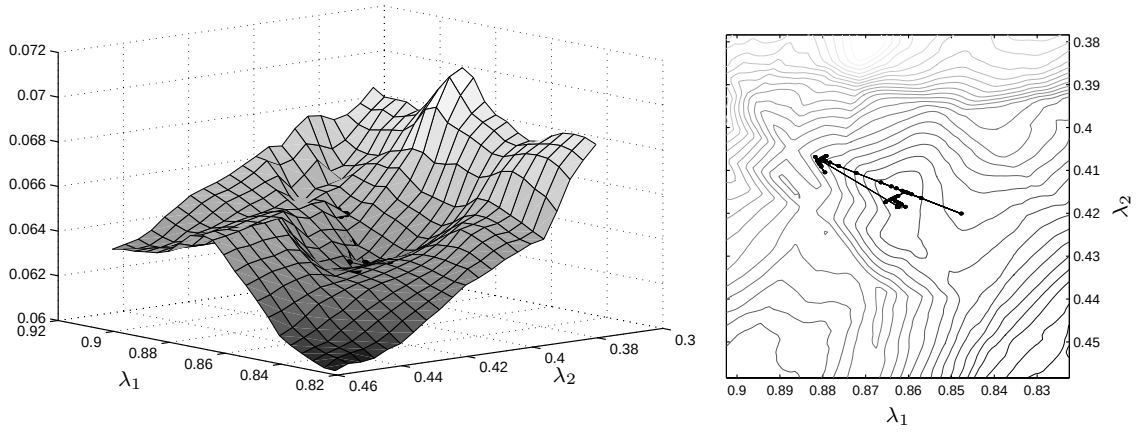
(*a*) Decreasing (*b*) Increasing

(*c*) Random (*d*) Euclidean

Figure 7.3: Ridge parameters after optimization for four different initial metric weight configurations. In each figure the initial weights are shown by the line and the weights after optimization are shown by the stem plot.

It is equally important that the ridge parameters be initialized properly. Since the later inputs should have less influence on the local linear model, the ridge parameters should be larger for these inputs. For the results reported here a linearly increasing function was used to initialize these parameters. Figure 7.3 shows the initial parameters and their values after optimization. Note that the optimization only decreased the penalty for the first few inputs, which confirms that these are the most important inputs for building the local linear models.

7.4 Multi-Step Cross Validation



(a) (b)

Figure 7.4: Example of a local minima near the initial values for vectored ridge regression and $n_a = 5$. (a) A slice of the cross-validation error surface for the two leading metric weights. All other coefficients were kept fixed. (b) Convergence of the optimization algorithm to the local minimum, located at the center of the figure.

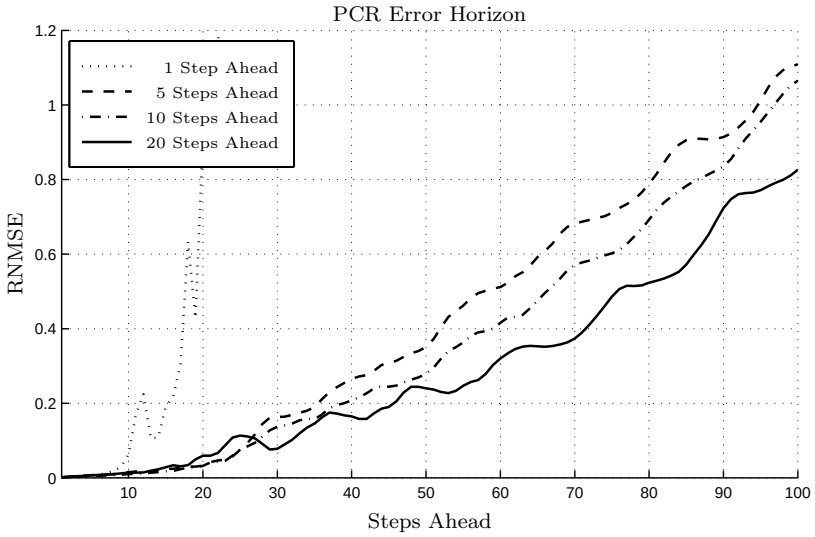
The previous chapter discussed the motivation for optimizing models to minimize the multi-step cross-validation error (MSCVE),

$$\overline{\mathrm{CVE}}(\gamma) \triangleq \frac{1}{n_a n_c} \sum_{i=1}^{n_c} \sum_{j=1}^{n_a} \left(y_{c(i)+j} - \hat{g}_\gamma^-(x_{c(i)+j-1}) \right)^2 \tag{7.2}$$

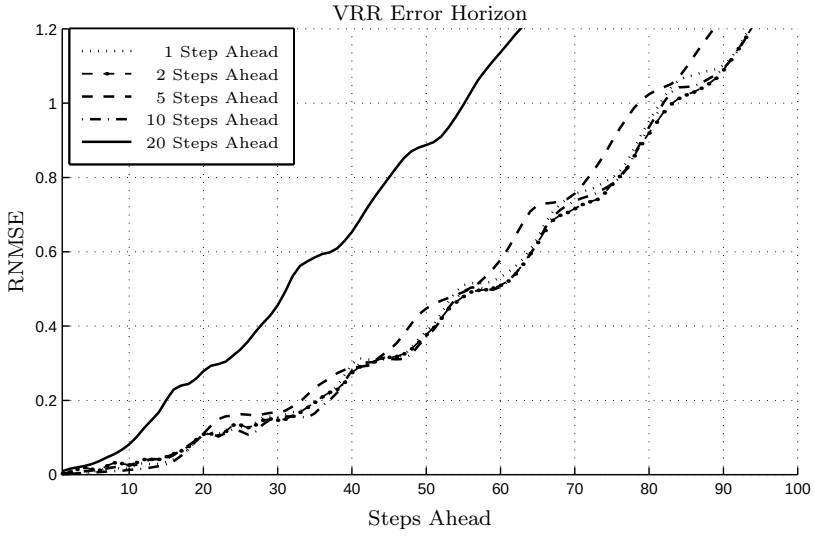
where n_c is the number of cross-validation points and n_a is the number of steps ahead. To illustrate the importance of this technique the prediction horizon for the Lorenz time series is shown for several different values of n_a in Figure 7.5.

As shown in the figure, increasing the value of n_a consistently increased the long term accuracy for PCR. This is because the one-step CVE does not account for estimation errors in the input vector and is thereby less sensitive to ill-conditioning. As this model is iterated and errors accumulate in the input vector, the errors are amplified by the ill-conditioning and the model output diverges. The multi-step cross-validation error (MSCVE) is able to account for these errors and consequently the model is able to generate multi-step predictions more accurately.

Performance for VRR decreases if n_a is increased too much because the number of local minima increases and the optimization converges to a sub-optimal solution more readily.



(*a*)



(*b*)

Figure 7.5: The error horizons for PCR and VRR applied to the Lorenz time series. The legend indicates how many steps ahead (n_a) the cross-validation error was calculated during parameter optimization.

This is illustrated by Figure 7.5, which shows little difference in the models optimized with $n_a = 1, 2, 5$ and 10, but for $n_a = 20$ the predictions were significantly worse. Figure 7.4 illustrates the convergence of the algorithm to a shallow local minimum that is very close to the initial parameter values for $n_a = 5$.

7.5 Time Series Prediction Examples

This section gives some examples of PCR and VRR applied to various time series. For each case, plots of the error horizon are given to compare the performance of PCR with VRR and to show the effect of minimizing the multi-step cross-validation error (MSCVE) for various values of steps ahead. Each time series also includes plots of typical predictions to give the reader an intuitive sense of how accurate these methods are.

7.5.1 Lorenz

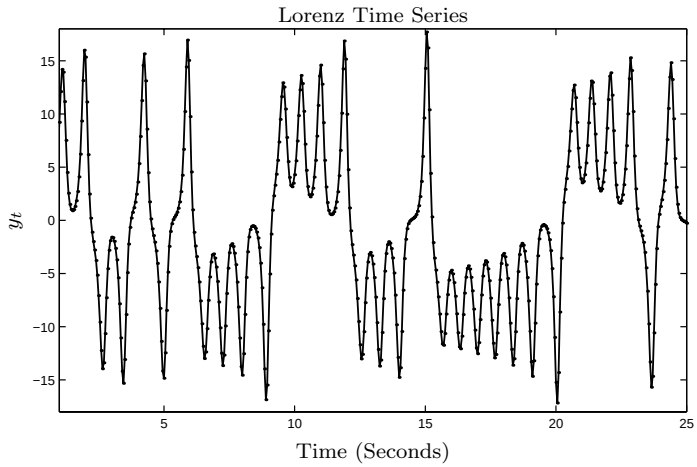


Figure 7.6: The first 500 points in the Lorenz time series.

The Lorenz system is an idealized model of fluid motion between a hot surface and a cool surface. It is described by the following nonlinear ordinary differential equations

$$\begin{aligned} \dot{z}_{t,1} &= \sigma(z_{t,2} - z_{t,1}), \\ \dot{z}_{t,2} &= r z_{t,1} - z_{t,2} - z_{t,1} z_{t,3}, \\ \dot{z}_{t,3} &= z_{t,1} z_{t,2} - b z_{t,3}, \\ y_t &= z_{t,3}, \end{aligned}$$

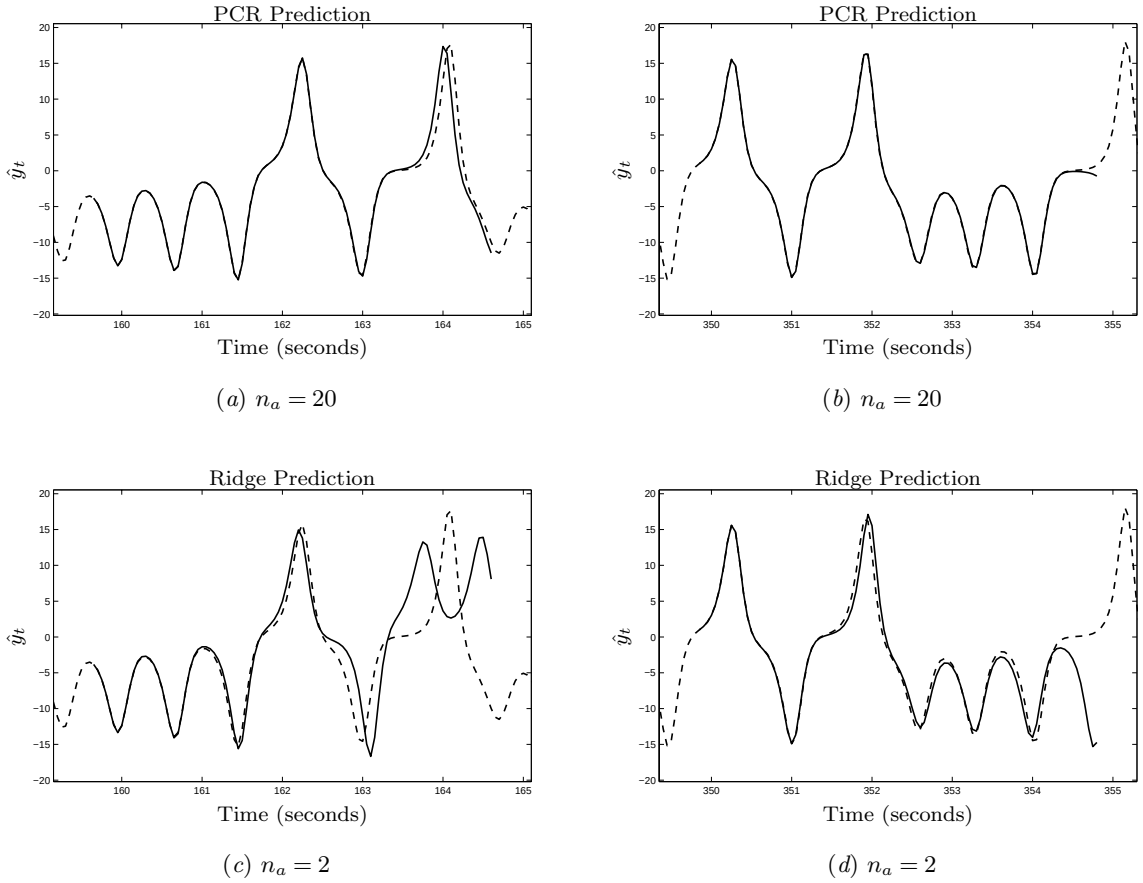


Figure 7.7: Prediction examples for Lorenz time series. The predicted series is shown by the solid line and the real series is shown by the broken line.

where $z_t \in \mathbb{R}^3$ is the state vector. The system is usually evaluated using the parameter values $\sigma = 10$, $r = 28$, and $b = \frac{8}{3}$.

This system played an integral role in the discovery of chaotic systems in the 1950s by Edward Lorenz, a meteorologist at MIT who was interested in using the model for weather forecasting [140, pp. 359–70].

Figure 7.6 shows 500 points in the Lorenz time series sampled every 0.05 seconds[1]. The time series consists of growing oscillations centered at approximately 9 and -9. Once the amplitude of the oscillations grows large enough, the system experiences a collapse and rapidly shifts to a different state. The collapses are difficult to predict accurately, which has motivated many researchers to use this time series to demonstrate their methods of prediction.

[1]This time series was generated using MATLAB's `ode43` function.

The prediction horizon for this time series is shown in Figure 7.5 and was discussed in Section 7.4. Figure 7.7 shows some of the predictions generated by PCR for $n_a = 20$ and by VRR for $n_a = 2$.

7.5.2 Mackey Glass

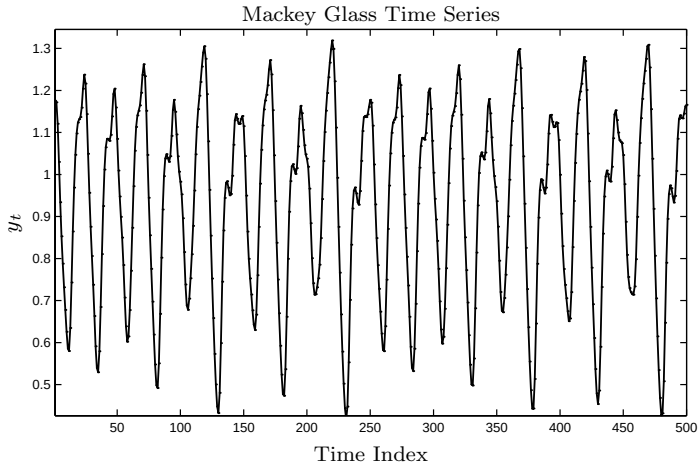


Figure 7.8: The first 500 points in Mackey Glass time series.

The Mackey-Glass time series has also played an important role in research of chaotic time series. The time series is described by

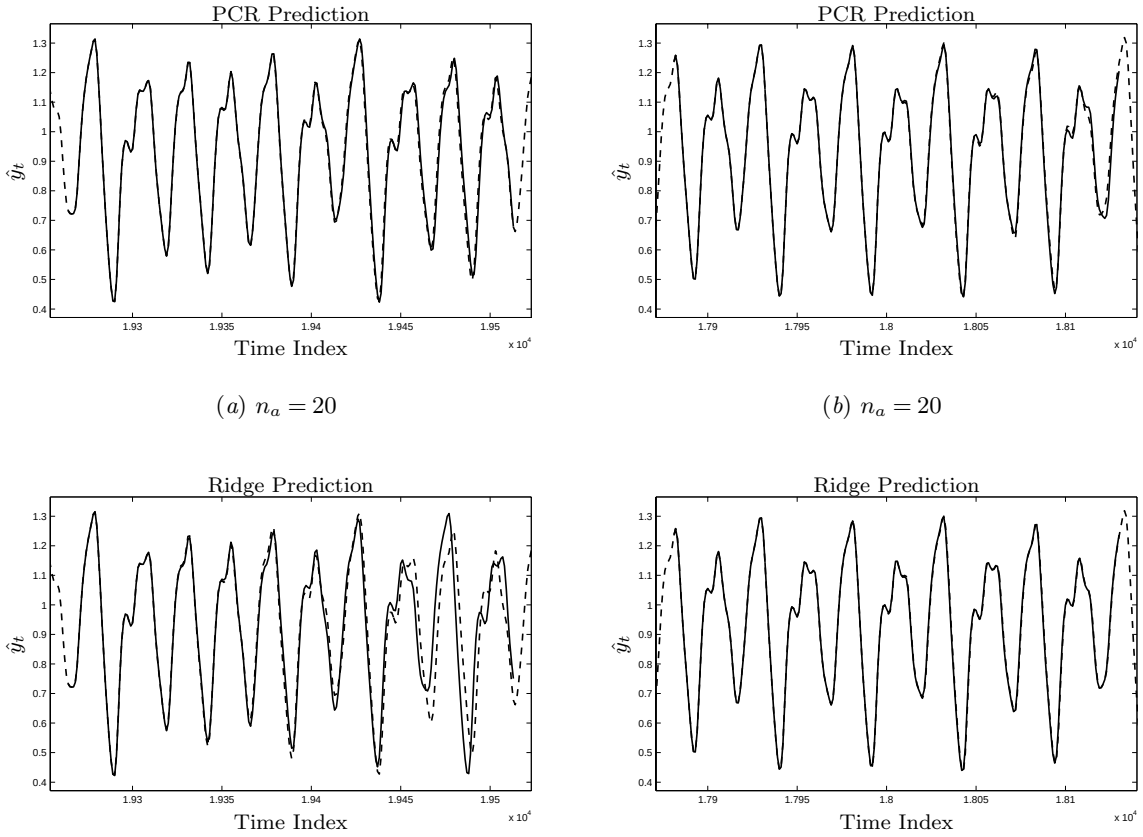
$$\dot{z}_t = \frac{0.2 z_{t-\Delta}}{1 + (z_{t-\Delta})^{10}} + 0.1 z_t,$$

where the Δ is usually set to 17. Figure 7.8 shows 500 points of the Mackey-Glass time series[2].

PCR and VRR models were constructed using the first $2,000$ points in the series. Figure 7.10 shows the error horizons for PCR and VRR. As for the Lorenz time series, the multi-step optimization produced a much better model than the one-step optimization for PCR.

In this case, VRR also benefited from multi-step optimization. The performance was much better for $n_a = 5, 10$, and 20 than for $n_a = 1$ or 2. As with the Lorenz time series, the overall performance of PCR was significantly better than VRR. Figure 7.9 gives some examples of the predictions generated by the models.

[2]This time series was simulated in Matlab by code provided by E. Wan. The code was publicly available at *http://www.ece.ogi.edu/~ericwan/data.html* at the time of writing.



(*a*) $n_a = 20$ (*b*) $n_a = 20$

(*c*) $n_a = 5$ (*d*) $n_a = 5$

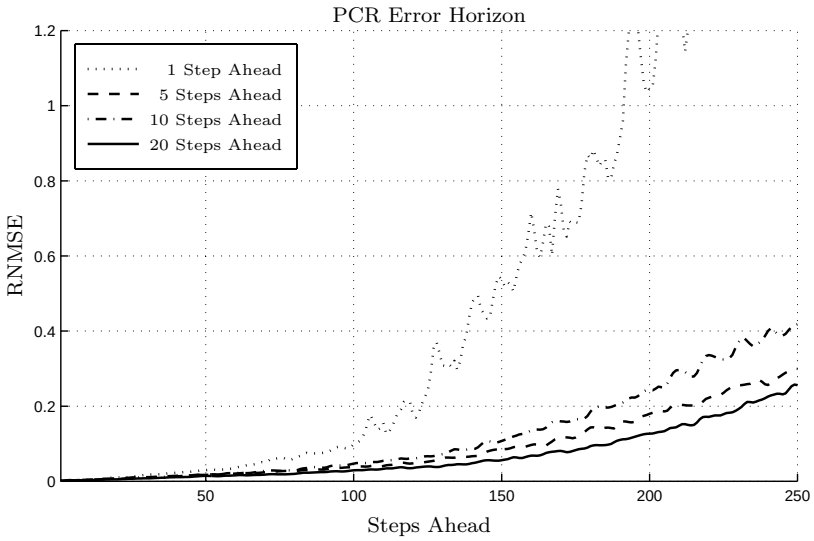
Figure 7.9: Prediction examples for Mackey Glass time series. The predicted series is shown by the solid line and the real series is shown by the broken line.

7.5.3 The Santa Fe Competition

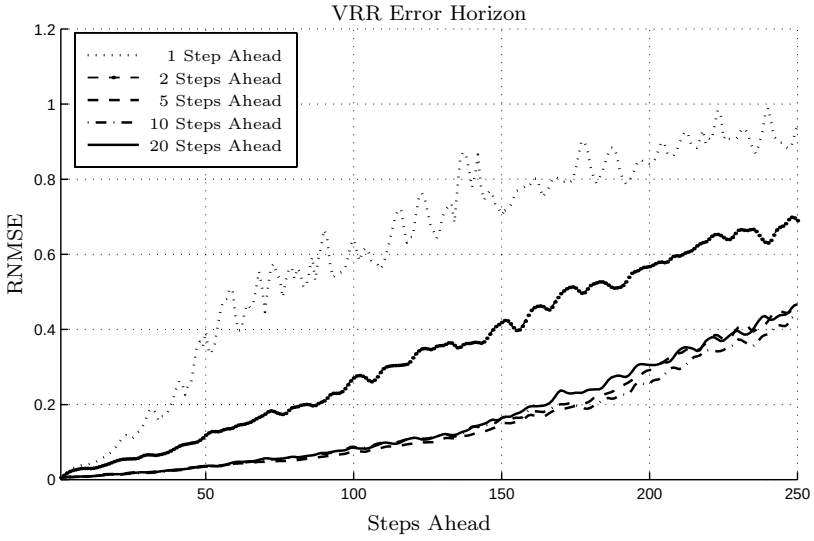
The Santa Fe time series prediction competition data set[3] is shown in Figure 7.11. This time series was generated by recording the light intensity from a NH_3 far-infrared laser and is thoroughly described in [141].

This is an especially difficult prediction problem because the time series is short, 1,000 points, it contains significant quantization noise (eight bits of resolution), it is undersampled, and it only contains three examples of the growing oscillations collapsing. During the competition entrants were asked to predict the next 100 points in the series. After the competition, the two best methods generated by Wan [38] and Sauer [37] were used to

[3]Available at *http://www.stern.nyu.edu/∼aweigend/Time-Series/SantaFe.html* at the time of writing.



(*a*)



(*b*)

Figure 7.10: The error horizons for PCR and VRR applied to the Mackey Glass time series. The legend indicates how many steps ahead (n_a) the cross-validation error was calculated during parameter optimization.

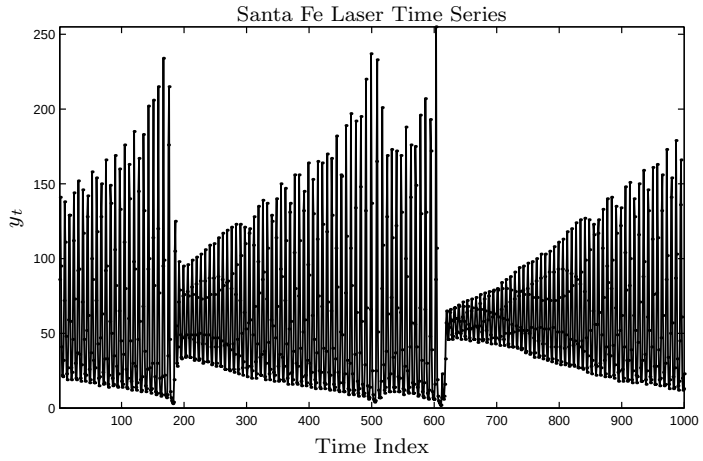


Figure 7.11: The Santa Fe laser time series.

make four other 100-point predictions at later sections of the time series using the same model used to generate their entries.

Start	PCR	VRR	Sauer	Wan
1002	0.037	0.508	0.077	0.027
2182	0.058	0.061	0.174	0.065
3872	0.521	0.061	0.183	0.487
4002	0.004	0.001	0.006	0.023
5182	0.029	0.262	0.111	0.160

Table 7.2: Performance statistics for Santa Fe laser time series using the first 1,000 points to build the models.

This analysis was repeated using the new methods described here[4].

Prior to building each model the time series was upsampled by eight to compensate for the sparse sampling. The range of nearest neighbors was also reduced to $k_{\min} = 2$ and $k_{\max} = 5$ since the series only contained a few examples of collapses. The results using $n_a = 10$ for PCR and $n_a = 2$ for VRR are given in Table 7.2.

The five predictions generated for each model are shown in Figure 7.12. In all cases, the models accurately predicted a collapse near the time of the actual collapse and in almost all cases the magnitude of the collapse was accurately predicted.

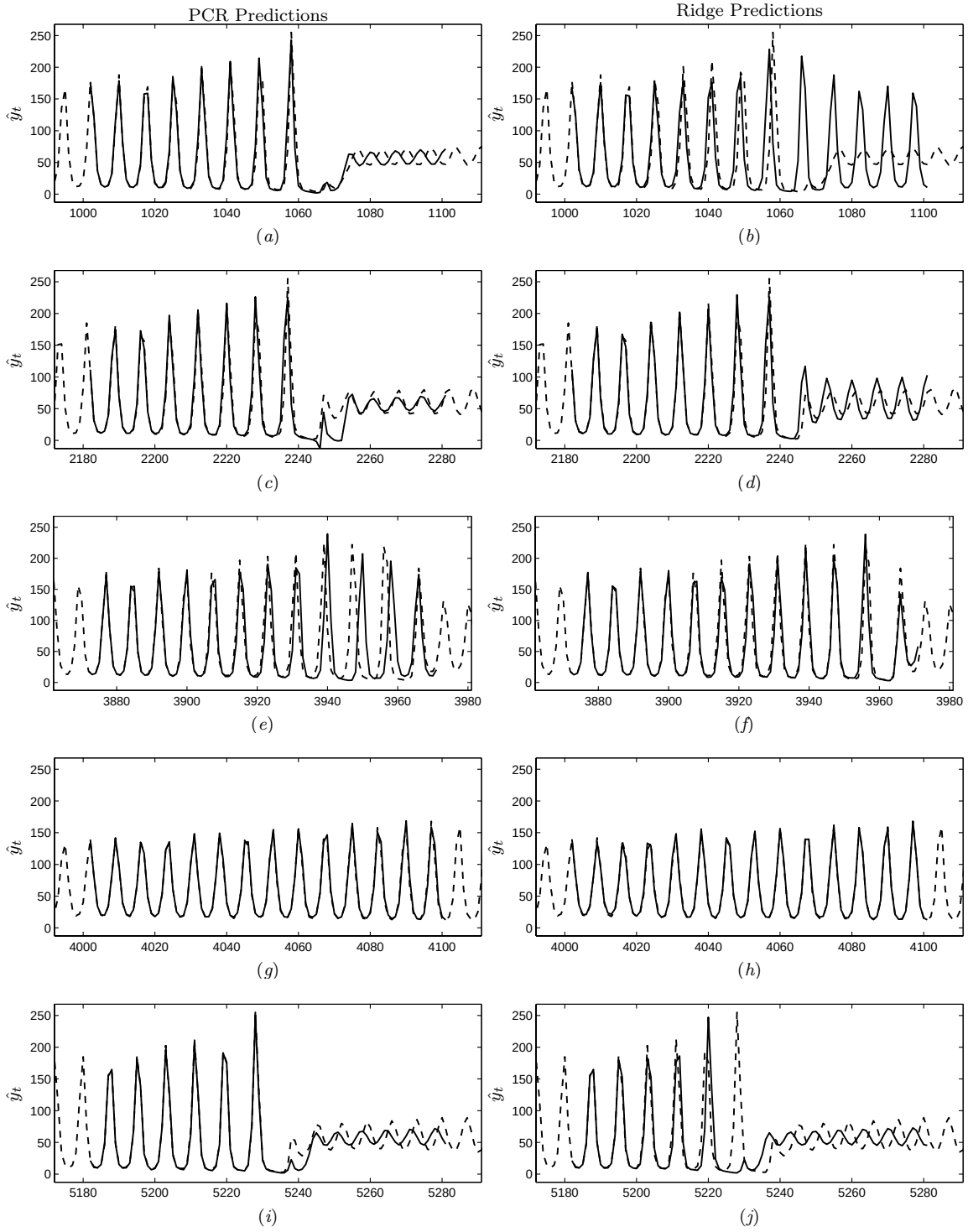


Figure 7.12: Prediction examples for Santa Fe laser time series. The true time series is shown by the broken line and the predicted time series is shown by the solid line.

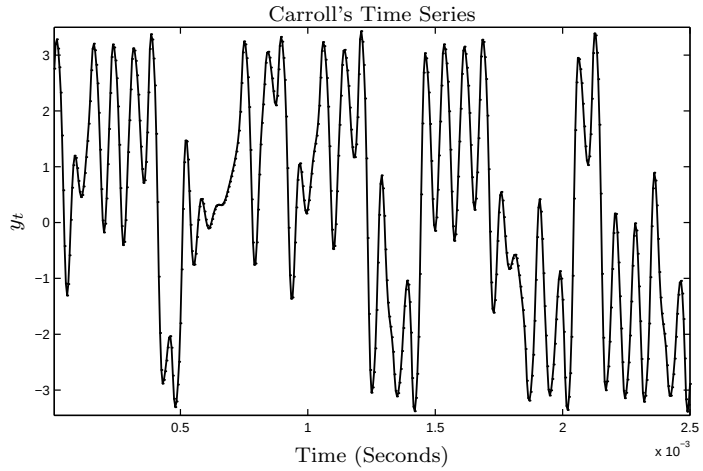


Figure 7.13: The first 500 points in Carroll's time series, generated by a nonlinear electric circuit.

7.5.4 Carroll's Circuit

Figure 7.13 shows 500 points from a chaotic time series recorded from an electric circuit[5]. The circuit is similar to the one described in [142].

The PCR and VRR were constructed using the first 3,000 points in the series. Figure 7.14 shows the error horizons for PCR and VRR. In this case, there was little difference among the models that minimized the multi-step cross-validation error, though the model optimized to predict one step ahead was notably worse for PCR. The PCR models were slightly better than the VRR models.

Figure 7.15 gives some examples of the predictions generated by the models.

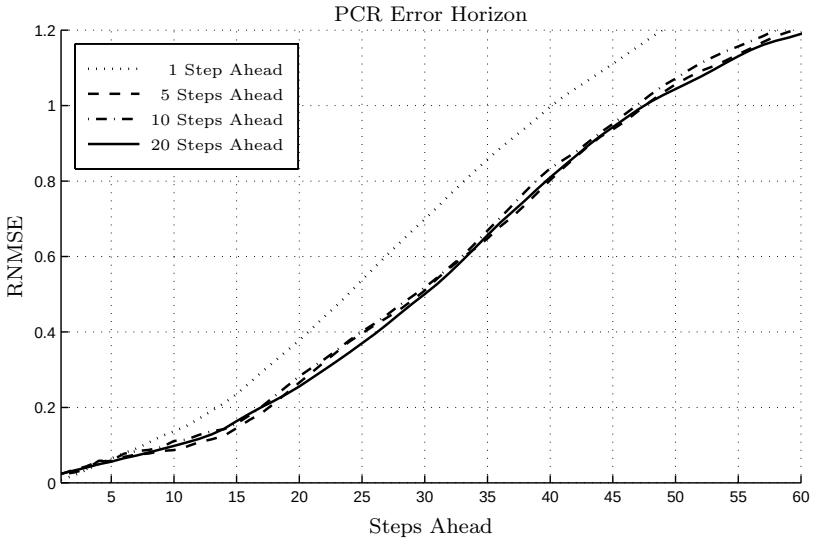
7.5.5 Weeks' Rotating Annulus

Figure 7.16 shows 500 points from a time series taken from a rotating annulus. The time series was recorded from a hot film probe used to measure the velocity of fluid flow at a fixed point. This experiment was conducted to study weather patterns in the Earth's atmosphere [143][6].

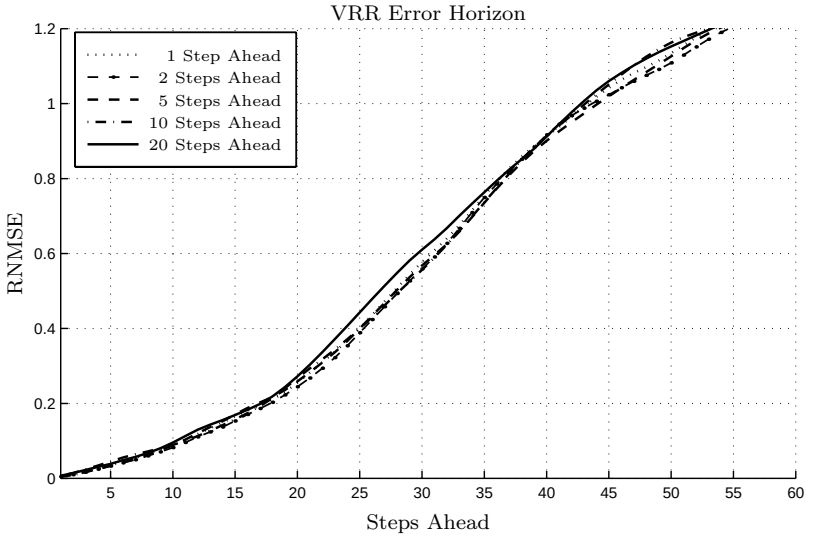
[4]A careful examination of the predictions generated by Wan [38] and Sauer [37] indicates that their predictions were offset by one. Wan started the predictions at 1001, 2181, 3871, 4001, and 5181 whereas Sauer started the predictions at 1002, 2182, 3872, 4002, and 5182. The predictions generated by PCR and VRR were started at the same times as Sauer's predictions.

[5]This data set was provided personally by T. Carroll.

[6]This data was publicly available at *http://∼glinda.lrsm.upenn.edu/∼weeks/∼research/∼tseries1.html* at the time of writing .

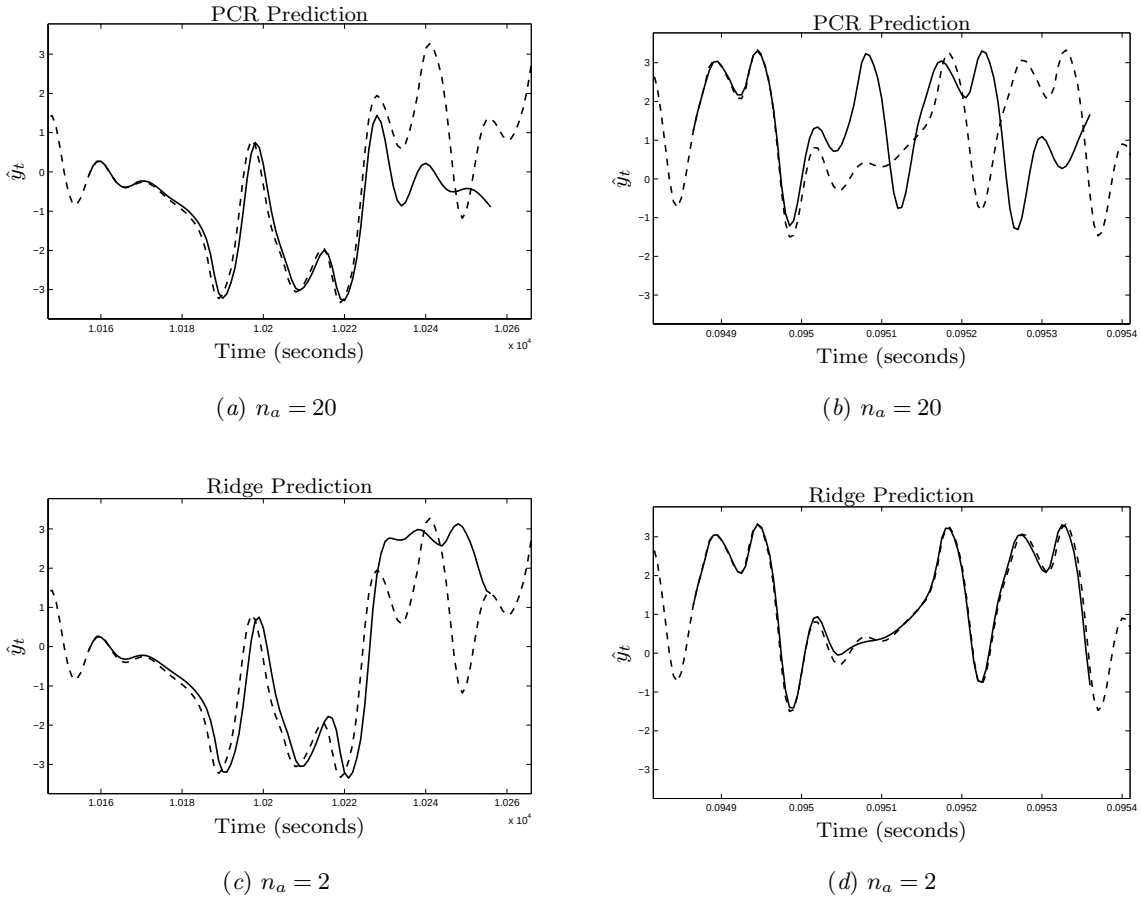


(*a*)



(*b*)

Figure 7.14: The error horizons for PCR and VRR applied to Carroll's time series. The legend indicates how many steps ahead (n_a) the cross-validation error was calculated during parameter optimization.



(a) $n_a = 20$ (b) $n_a = 20$

(c) $n_a = 2$ (d) $n_a = 2$

Figure 7.15: Prediction examples for Carroll time series. The predicted series is shown by the solid line and the real series is shown by the broken line.

PCR and VRR models were constructed using the first $5,000$ points in the series. Figure 7.17 shows the error horizons for PCR and VRR. The results were similar to those of Carroll's circuit; there was little difference among the models that minimized the multi-step cross-validation error, though the model optimized to predict one step ahead was notably worse for PCR. The PCR models were slightly better than the VRR models.

Figure 7.18 gives some examples of the predictions generated by the models.

7.6 Summary

It is difficult to compare the new optimization methods described in this work with previous methods because previously researchers did not use optimization algorithms to find values for the critical model parameters. Rather, researchers chose parameter values based on their own experience and expertise. However, a comparative study is not necessary since

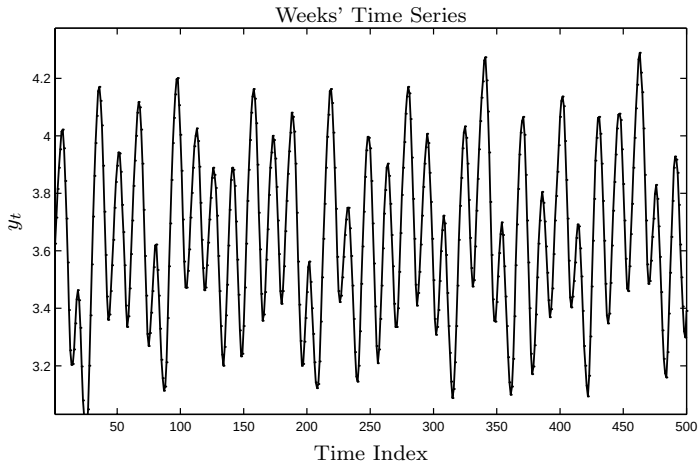


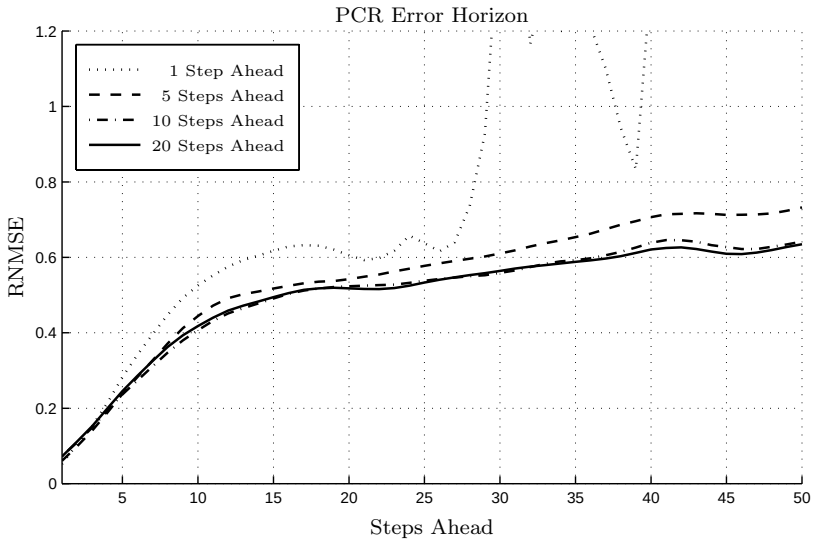
Figure 7.16: The first 500 points in Weeks' time series, generated from a rotating annulus.

the new methods use the parameter values selected by the user as a starting point and can only improve performance, as measured by the cross-validation error.

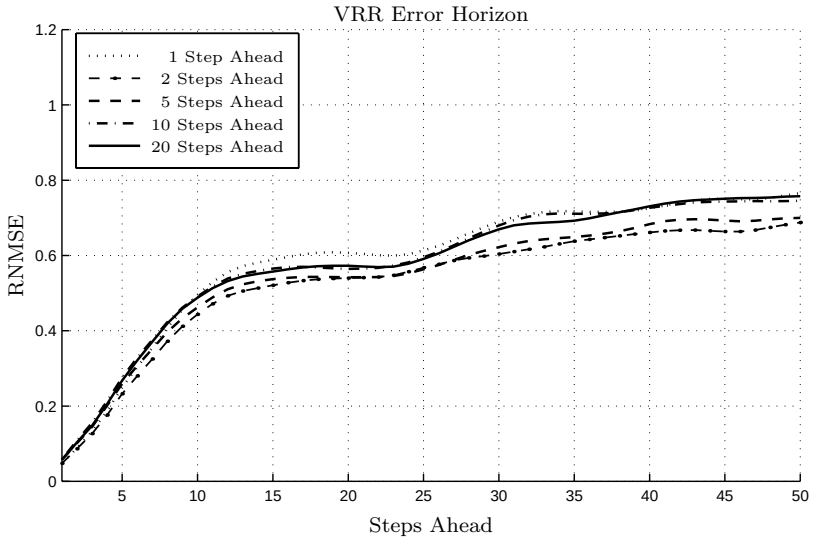
Instead, this chapter focused on two aspects of the new ideas presented in this work. First, it compared the performance of vectored ridge regression (VRR) with principal components regression (PCR) on a variety of chaotic time series prediction problems. Both of these forms of regularization are new generalizations that supersede their predecessors and, if parameter values are chosen appropriately, can only improve performance. On all of the time series described here, PCR performed significantly better than VRR.

Second, this chapter examined the effect of minimizing the multi-step cross-validation error (MSCVE) for various values of the number of steps ahead. This was a key innovation designed specifically for chaotic time series (Chapter 6). On all of the time series considered in this chapter, the optimization of MSCVE substantially improved the performance of principal components regression (PCR) as compared to the one-step cross-validation error. Generally, as the number of steps ahead was increased, the performance of PCR improved or stayed the same.

The improvement was less dramatic for vectored ridge regression (VRR) because the multi-step cross-validation error contains more local minima than the one-step cross-validation error. The optimization of VRR is especially susceptible to local minima because it uses gradient-based optimization, which tries to find the nearest local minima in as few steps as possible. PCR is more tolerant of local minima because it uses a semi-global line search for the real-valued parameters and an exhaustive search for the integer-valued parameters.

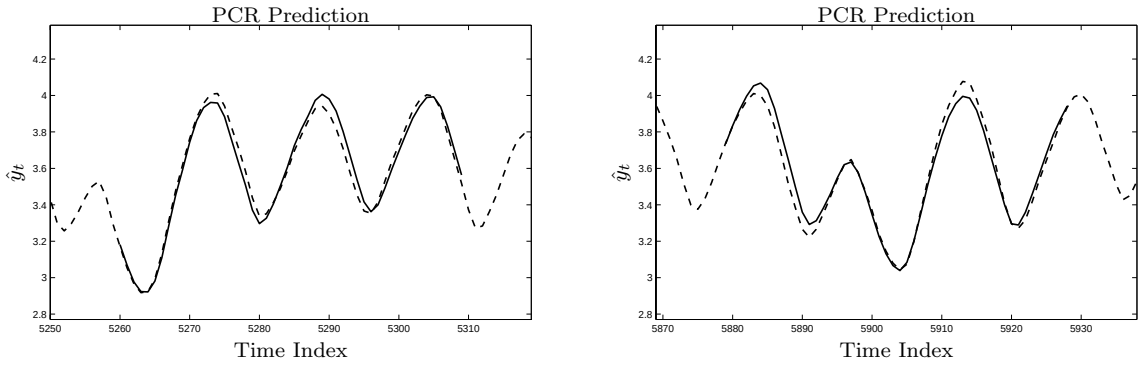


(*a*)

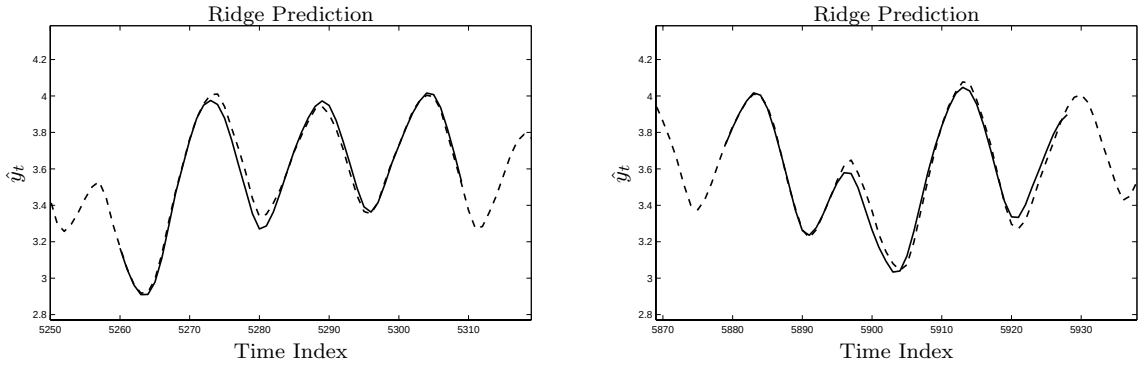


(*b*)

Figure 7.17: The error horizons for PCR and VRR applied to Weeks' time series. The legend indicates how many steps ahead (n_a) the cross-validation error was calculated during parameter optimization.



(*a*) $n_a = 20$ (*b*) $n_a = 20$



(*c*) $n_a = 2$ (*d*) $n_a = 2$

Figure 7.18: Prediction examples for Weeks' time series. The predicted series is shown by the solid line and the real series is shown by the broken line.

Chapter 8

Conclusion

The concept is interesting and well-formed, but in order to earn better than a C*, the idea must be feasible.*
—A Yale University management professor in response to Fred Smith's paper proposing reliable overnight delivery service. (Smith went on to found Federal Express.)

This chapter is organized as follows. Section 8.1 briefly reviews the preceding chapters and Section 8.2 suggests several directions for future research.

8.1 Summary

This work addressed three fundamental problems for local modeling: the nonlinear modeling problem, the nearest neighbors problem, and the time series prediction problem. Each of these problems occurs in many applications, as described in Chapter 1.

Chapter 3 described two new fast nearest neighbor algorithms, principal axis trees (PAT) and depth-only principal axis trees (DOPAT). Both of these algorithms were shown to perform well compared to leading techniques on a variety of benchmarks.

Chapter 4 introduced ridge regression and principal components regression, two popular methods of coping with an ill-conditioned matrix inverse that occurs when constructing local linear models. Both of these methods were generalized for the weighted squared error penalty function. Ridge regression was also generalized to have a separate penalty for each input variable and soft thresholds were used for principal components regression to diminish the effect of small singular values.

Chapter 5 described a method of optimizing local models using a generalization of the cyclic coordinate method. This algorithm optimizes some of the critical parameter values one at a time using an exhaustive search for integer-valued parameters and a semi-global line search for real-valued parameters. The parameters for which the gradient can be calculated are optimized using a standard gradient-based optimization algorithm.

Chapter 6 modified these algorithms in several important ways to increase efficiency and accuracy of local models for the time series prediction problem. Most importantly, these algorithms were modified to minimize the multi-step cross-validation error (MSCVE) which ensures the local models account for the errors that occur with iteration. The importance of this modification was illustrated in Chapter 7, which demonstrated the performance of optimized local models on a variety of time series prediction problems.

8.2 Future Research

This section suggests several ideas for further research that may extend and improve the new methods described in this work.

8.2.1 Non-Chaotic Time Series

The discussion of time series prediction was limited to chaotic time series because Takens' theorem provides a strong theoretical foundation for this approach, to limit the scope of this work to a reasonable length, and because chaotic time series have proven to be difficult to predict in practice. However, there is nothing that prevents this approach from being used to forecast other types of time series. It would be interesting to compare these new techniques with existing methods on some of the more traditional time series prediction problems in the areas of economics, agriculture, demographics, climate trends, epidemiology, and weather forecasting.

8.2.2 Local Minima

One of the most elegant properties of the optimization algorithms developed here is that they can only improve the model accuracy compared to the initial parameter values provided by the user. However, these algorithms can get stuck in shallow local minima after only modestly improving the model performance.

Principal components regression (PCR) optimization for time series prediction is less susceptible to this problem than vectored ridge regression (VRR) because all of the parameters are optimized by either an exhaustive search or a semi-global line search. This may partially explain why PCR consistently out-performed VRR on the prediction problems discussed in Chapter 7.

There are a variety of techniques that could be used to help prevent VRR from falling into shallow local minima. Some techniques, such as genetic algorithms, simulated annealing, and evolutionary programming, are designed specifically for this purpose. However, each of these techniques are computationally demanding and would require much more time for optimization.

As demonstrated in Chapter 7, local averaging is much less susceptible to shallow local minima. It may be best to use the metric weights after optimization with local averaging as the initial values for VRR. It may also improve the model accuracy if VRR is optimized to predict one step ahead, then two steps ahead, and so on. This is a difficult, but important, problem that deserves further consideration.

8.2.3 Local Optimization

In this work all of the model parameters were held constant during predictions. Some researchers have suggested optimizing the model parameters using only a local neighborhood of points [121, 144, 145]. This approach is difficult because it is hard to estimate the model accuracy accurately using only the neighboring points and it requires additional computation for the optimization after the input vectors are available. Nonetheless, the initial results reported by others are encouraging and it should be possible to generalize the model structures and algorithms described here for local optimization.

8.2.4 Multiple Iterative Models

Chapter 6 discussed the importance of using the multi-step cross-validation error to account for the errors in model input vectors. An alternative to coping with this problem is to use a different model to predict each step ahead. This idea is illustrated in Figure 8.1.

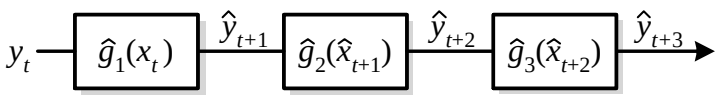


Figure 8.1: Multiple iterative models.

Multiple iterative models are more general than a single model. If the models are optimized appropriately, they can only improve performance as compared to single iterative models.

Initial results indicate that, after optimization, the multiple iterative models are not as accurate as single iterative models. This is probably because the optimization of each model does not take into account the effect it will have on the predictions generated by the other models. Nonetheless, this approach has much potential for further improvement.

8.3 Concluding Remarks

The methods developed here were greatly influenced by processing power available in modern computers. The model structure was designed to be flexible enough to generate accurate long term forecasts of chaotic time series and yet efficient enough that the parameters could be optimized in less than two hours.

The optimization algorithms were specifically designed to meet two goals. First, they should only improve model performance from that of the initial parameter values provided by the user. Cross-validation error played a vital role in their development because it is resistant to the problem of overfitting and it can be calculated efficiently for local models.

Second, the algorithms should relieve the user of responsibility for picking critical values for model parameters; this was replaced with the new responsibility of picking values that control the amount of time required for model optimization. This enables the user to scale the thoroughness of the searches, and thereby the model accuracy, to suit the time constraints and computational resources of the application at hand. This also ensures the algorithms will become more powerful and can be applied to larger data sets as the power of computers continues to increase.

Appendix A

Vectored Ridge Regression Gradients

This "telephone" has too many shortcomings to be seriously considered as a means of communication. The device is inherently of no value to us.
—Western Union internal memo, 1876.

This appendix contains several mathematical derivations for vectored ridge regression (VRR), introduced in Chapter 4. Section A.1 briefly reviews the notation used for local linear models. Section A.2 derives the model coefficients that minimize the weighted, penalized sum of squared errors, a generalization of the ordinary least squares solution.

The remainder of this appendix is devoted to finding the gradient of the model output with respect to the weighted Euclidean metric parameters (Section A.3), the model inputs (Section A.4), and the ridge regression parameters (Section A.5). The ability to efficiently calculate these gradients plays a vital role in parameter optimization, described in Chapters 5 and 6.

A.1 Model Description

The matrix A is used to hold each of the k neighboring points and the vector b is used to store the target value for each of the k points,

$$A \triangleq \begin{bmatrix} x_1^{\mathrm{T}} & 1 \\ x_2^{\mathrm{T}} & 1 \\ \vdots & \\ x_k^{\mathrm{T}} & 1 \end{bmatrix}, \qquad b \triangleq \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix}.$$

Each row in the matrix A is composed of a vector of observed variables with an appended value of one.

For a query vector, q, the model output is given by

$$\begin{aligned} \hat{y} &= [q^{\mathrm{T}} \;\; 1]\, \nu, \\ &= \nu_{n_d+1} + \sum_{i=1}^{n_d} q_i \nu_i \,, \end{aligned}$$

where q, $x_i \in \mathbb{R}^{n_d}$. In this appendix $\tilde{q}$ is used to denote the vector q appended with a constant value of one,

$$\tilde{q}^T \triangleq [q^{\mathrm{T}} \;\; 1].$$

A.2 Weighted Vectored Ridge Regression Solution

The linear model coefficients are usually chosen to minimize a user-specified error criterion. There are many types of error measures that could be used but most users prefer the sum of squared errors because a deterministic solution exists that can be calculated efficiently. The deterministic solution that minimizes the *weighted* sum of squared errors, a more general error measure, is derived in this section. This error measure is given by

$$\begin{aligned} \mathrm{P}_{\mathrm{WVRR}}(\nu) &\triangleq \sum_{i=1}^{k} w_i^2 \left(y_i - \hat{y}(x_i)\right)^2 + \sum_{i=1}^{n_d+1} r_i^2 \nu_i^2, \\ &= (b - A\nu)^{\mathrm{T}} W^2 (b - A\nu) + \nu^{\mathrm{T}} R^2 \nu, \\ &= b^{\mathrm{T}} W^2 b - \nu^{\mathrm{T}} A^{\mathrm{T}} W^2 b - b^{\mathrm{T}} W^2 A\nu + \nu^{\mathrm{T}} A^{\mathrm{T}} W^2 A\nu + \nu^{\mathrm{T}} R^2 \nu, \end{aligned}$$

where the matrices $W \in \mathbb{R}^{k\times k}$ and $R \in \mathbb{R}^{n_d \times n_d}$ are diagonal matrices of the vectors w and r,

$$W \triangleq \operatorname{diag}(w), \qquad R \triangleq \operatorname{diag}(r),$$
$$= \begin{bmatrix} w_1 & 0 & \cdots & 0 \\ 0 & w_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & w_k \end{bmatrix}, \qquad = \begin{bmatrix} r_1 & 0 & \cdots & 0 \\ 0 & r_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & r_{n_d} \end{bmatrix},$$

and W^2 denotes the product of a symmetric matrix with itself, $W^2 \triangleq W^{\mathrm{T}}W = WW^{\mathrm{T}}$.

Since $\mathrm{P}_{\mathrm{WVRR}}(\nu)$ is quadratic in ν there can only be one extremum, which is the value of ν for which the gradient of $\mathrm{P}_{\mathrm{WVRR}}(\nu)$ is equal to zero. The gradient is given by

$$\nabla_\nu \mathrm{P}_{\mathrm{WVRR}}(\nu) = -2A^{\mathrm{T}}W^2 b + 2A^{\mathrm{T}}W^2 A\nu + 2R^2\nu.$$

Setting this equation equal to zero and solving for ν yields

$$\nu \;=\; \left(A^T W^2 A + R^2\right)^{-1} A^{\mathrm{T}} W^2 b. \tag{A.1}$$

It is not difficult to show that the extremum found by solving $\nabla_\nu \mathrm{P}_{\mathrm{WVRR}} = 0$ is the value that minimizes $\mathrm{P}_{\mathrm{WVRR}}(\nu)$. Since the Hessian,

$$\nabla^2_\nu \mathrm{P}_{\mathrm{WVRR}}(\nu) = 2A^{\mathrm{T}}A + 2R^2,$$

is positive definite for any matrices A and R with independent columns [42, p. 333], $\mathrm{P}_{\mathrm{WVRR}}(\nu)$ is a strictly convex function of ν [43, Theorem 3.3.8]. Therefore, the value for ν given in Equation A.1 minimizes $\mathrm{P}_{\mathrm{WVRR}}(\nu)$ [43, Theorem 3.4.3, Corollary 1].

The output of the optimal model for a query q is given by

$$\begin{aligned} \hat{y} \;&=\; \tilde{q}^{\mathrm{T}}\nu, \\ &=\; \tilde{q}^{\mathrm{T}} \left(A^T W^2 A + R^2\right)^{-1} A^{\mathrm{T}} W^2 b. \end{aligned} \tag{A.2}$$

A.3 Metric Gradient

This section derives the gradient of the model output with respect to the metric parameters, $\nabla_\lambda \hat{y}$. The model output, $\hat{y}$ depends on the metric parameters λ only indirectly through

the penalty weights, w, as shown by Equation A.2. Thus, the jth element of $\nabla_\lambda \hat{y}$ can be written as

$$\frac{\partial \hat{y}}{\partial \lambda_j} = \sum_{i=1}^{k} \frac{\partial \hat{y}}{\partial w_i} \frac{\partial w_i}{\partial \lambda_j}.$$

Each element of the gradient can be combined into vector form to give

$$\nabla_\lambda \hat{y} = \frac{\partial w}{\partial \lambda} \left(\nabla_w \hat{y} \right), \tag{A.3}$$

where $\nabla_w \hat{y}$ is the gradient of the model output with respect to the penalty weights and $\frac{\partial w}{\partial \lambda} \in \mathbb{R}^{k \times n_d}$ is the Jacobian of the penalty weights with respect to the metric parameters.

Section A.3.1 derives the solution for $\nabla_w \hat{y}$, Section A.3.2 derives the solution for $\frac{\partial w}{\partial \lambda}$, and Section A.3.3 gives an algorithm that describes how to compute $\nabla_\lambda \hat{y}$.

A.3.1 Penalty Weight Gradient

The derivation of $\nabla_w \hat{y}$ begins with the equation for the model output, which is repeated here for convenience,

$$\begin{aligned} \hat{y} &= \tilde{q}^{\mathrm{T}} \left(A^T W^2 A + R^2 \right)^{-1} A^{\mathrm{T}} W^2 b \\ &= \tilde{q}^{\mathrm{T}} S^{-1} A^{\mathrm{T}} W^2 b, \end{aligned}$$

where $S \triangleq A^T W^2 A + R^2$. The ith element of the gradient is given by,

$$\begin{aligned} \frac{\partial \hat{y}}{\partial w_i} &= \left(\nabla_w \hat{y} \right)_i, \\ &= \underbrace{\left(\frac{\partial}{\partial w_i} \tilde{q}^{\mathrm{T}} S^{-1} \right) A^{\mathrm{T}} W^2 b}_{①} + \underbrace{\tilde{q}^{\mathrm{T}} S^{-1} \left(\frac{\partial}{\partial w_i} A^2 W^2 b \right)}_{②}, \end{aligned} \tag{A.4}$$

where the second equation was obtained by the chain rule. The derivative of the first term in this equation can be solved for as follows,

$$\begin{aligned}
① &\triangleq \left(\frac{\partial}{\partial w_i}\tilde{q}^{\mathrm{T}}S^{-1}\right)A^{\mathrm{T}}W^2b, \\
&= \tilde{q}^{\mathrm{T}}\left(\frac{\partial}{\partial w_i}S^{-1}\right)A^{\mathrm{T}}W^2b, \\
&= -\tilde{q}^{\mathrm{T}}S^{-1}\left(\frac{\partial}{\partial w_i}S\right)S^{-1}A^{\mathrm{T}}W^2b, \\
&= -\tilde{q}^{\mathrm{T}}S^{-1}\left(\frac{\partial}{\partial w_i}S\right)\nu, \\
&= -\tilde{q}^{\mathrm{T}}S^{-1}\left(A^{\mathrm{T}}\frac{\partial}{\partial w_i}W^2A\right)\nu,
\end{aligned}$$

where the third step used the equation for a derivative of a matrix [146, p. 902–3],

$$\frac{d}{d\alpha}C^{-1} = -C^{-1}\left(\frac{d}{d\alpha}C\right)C^{-1}. \tag{A.5}$$

Since $W = \mathrm{diag}(w)$, $\frac{\partial}{\partial w_i}W^2$ can be written as $e_i 2w_i e_i^{\mathrm{T}}$ where e_i is defined as the ith column of the identity matrix, $e_i \triangleq [0\cdots1\cdots0]^{\mathrm{T}}$, a vector of zeros, except for the ith element, which is equal to one. The derivative is then given by

$$\begin{aligned}
① &= -\tilde{q}^{\mathrm{T}}S^{-1}A^{\mathrm{T}}\left(\frac{\partial}{\partial w_i}W^2\right)A\nu, \\
&= -\tilde{q}^{\mathrm{T}}S^{-1}A^{\mathrm{T}}\left(e_i 2w_i e_i^{\mathrm{T}}\right)A\nu, \\
&= \left(-2w_i\right)\left(\tilde{q}^{\mathrm{T}}S^{-1}A^{\mathrm{T}}e_i\right)\left(e_i^{\mathrm{T}}A\nu\right).
\end{aligned}$$

The parentheses are used to illustrate that the linear equation reduces to a product of three scalars.

The second term in Equation A.4 can solved for as follows:

$$\begin{aligned}
② &\triangleq \tilde{q}^{\mathrm{T}}S^{-1}\left(\frac{\partial}{\partial w_i}A^{\mathrm{T}}W^2b\right), \\
&= \tilde{q}^{\mathrm{T}}S^{-1}A^{\mathrm{T}}\left(\frac{\partial}{\partial w_i}W^2\right)b, \\
&= \tilde{q}^{\mathrm{T}}S^{-1}A^{\mathrm{T}}\left(e_i 2w_i e_i^{\mathrm{T}}\right)b, \\
&= \left(2w_i\right)\left(\tilde{q}^{\mathrm{T}}S^{-1}A^{\mathrm{T}}e_i\right)\left(e_i^{\mathrm{T}}b\right),
\end{aligned}$$

where parenthesis have been used again to illustrate that the linear equations reduce to a product of three scalars. Combining these equations yields,

$$\begin{aligned}\frac{\partial \hat{y}}{\partial w_i} &= ① + ②, \\ &= (-2w_i)\left(\tilde{q}^{\mathrm{T}} S^{-1} A^{\mathrm{T}} e_i\right)\left(e_i^{\mathrm{T}} A\nu\right) + (2w_i)\left(\tilde{q}^{\mathrm{T}} S^{-1} A^{\mathrm{T}} e_i\right)\left(e_i^{\mathrm{T}} b\right), \\ &= (2w_i)\left(\tilde{q}^{\mathrm{T}} S^{-1} A^{\mathrm{T}} e_i\right)\left(e_i^{\mathrm{T}} b - e_i^{\mathrm{T}} A\nu\right), \\ &= (2w_i)\left(e_i^{\mathrm{T}} A S^{-1} \tilde{q}\right)^{\mathrm{T}}\left(e_i^{\mathrm{T}}(b - A\nu)\right).\end{aligned}$$

Each element of the gradient can then be combined into a vector to give the final expression for the penalty weight gradient,

$$\nabla_w \hat{y} = 2\,\mathrm{diag}\left(A S^{-1} \tilde{q}\right)\mathrm{diag}\left(b - A\nu\right) w. \tag{A.6}$$

A.3.2 Metric Jacobian

This section derives a solution for the Jacobian of the penalty weight vector with respect to the metric weight vector, $\frac{\partial w}{\partial \lambda} \in \mathbb{R}^{k \times n_d}$.

The weights depend on the metric parameters indirectly through a local weighting function. Throughout this work the biweight function has been used, but it could be any smooth function. In this section, the weights are assumed to be a function of the distances as follows:

$$w_i \triangleq \mathrm{W}(\rho_i),$$

where $\rho_i \triangleq \frac{d_i^2}{d_{k+1}^2}$ and d_i is the weighted Euclidean distance between the ith nearest neighbor and the query point,

$$\begin{aligned} d_i^2 &\triangleq \mathrm{D}_{\mathrm{WE}}(q, x_i)^2, \\ &= \sum_{\ell=1}^{n_d} \lambda_\ell^2 \left(q_\ell - x_{i,\ell}\right)^2. \end{aligned}$$

The Jacobian's value in the jth row and the ith column is given by

$$\begin{aligned}
\frac{\partial w_i}{\partial \lambda_j} &= \frac{\partial}{\partial \lambda_j} \mathrm{W}(\rho_i), \\
&= \frac{d\,\mathrm{W}(\rho_i)}{d\rho_i} \frac{\partial \rho_i}{\partial \lambda_j}, \\
&= \frac{d\,\mathrm{W}(\rho_i)}{d\rho_i} \frac{\partial}{\partial \lambda_j} \left(\frac{d_i^2}{d_{k+1}^2} \right), \\
&= \frac{d\,\mathrm{W}(\rho_i)}{d\rho_i} \left(\frac{\partial d_i^2}{\partial \lambda_j} d_{k+1}^2 - d_i^2 \frac{\partial d_{k+1}^2}{\partial \lambda_j} \right) \left(\frac{1}{d_{k+1}^2} \right)^2 . \qquad \text{(A.7)}
\end{aligned}$$

If the biweight function is used[1], the equation for $\mathrm{W}(\rho_i)$ is given by

$$\begin{aligned}
\mathrm{W}(\rho_i) &= (1 - \rho_i), \\
&= \left(1 - \frac{d_i^2}{d_{k+1}^2} \right),
\end{aligned}$$

and the derivative of $W(\rho_i)$ is given by

$$\frac{d\,\mathrm{W}(\rho_i)}{d\rho_i} = -\rho_i.$$

The remaining derivatives of the Jacobian in Equation A.7 are readily obtained from the definition of the distance metric,

$$\begin{aligned}
\frac{\partial d_i^2}{\partial \lambda_j} &= \frac{\partial}{\partial \lambda_j} \left(\sum_{\ell=1}^{n_d} \lambda_\ell^2 (q_\ell - x_{i,\ell})^2 \right), \\
&= 2\lambda_j (q_j - x_{i,j})^2.
\end{aligned}$$

Once the Jacobian is calculated the gradient $\nabla_\lambda \hat{y}$ can be calculated by applying Equation A.3. The next section summarizes these steps in an algorithm.

A.3.3 Metric Gradient Algorithm

The process for calculating the gradient $\nabla_\lambda \hat{y}$ is summarized below. This algorithm assumes that the $k+1$ nearest neighbors have been found and that the local model has been constructed as described in Section A.1.

[1]The weights are squared in Equation A.1, so the square root of the biweight function is used here.

Algorithm A.1: Calculation of $\nabla_{\lambda}\hat{y}$

1. Calculate the penalty weight gradient:

   $\nabla_w \hat{y} = \mathrm{diag}(AS^{-1}\tilde{q})\,\mathrm{diag}(b - A\nu)2w.$

2. Calculate the derivative of the biweight function,

   $\dfrac{d\,\mathrm{W}(\rho_i)}{d\rho_i} = -\rho_i,$

   for each i, where

   $\rho_i \triangleq \dfrac{d_i^2}{d_{k+1}^2},$

   and d_i is the weighted Euclidean distance to the ith nearest neighbor.

3. Calculate the product of the Jacobian and $\nabla_w \hat{y}$.
   For $i = 1$ to n_d,

   3.1 Calculate

   $$\frac{\partial d_{k+1}^2}{\partial \lambda_i} := 2\lambda_i (q_i - x_{k+1,i})^2.$$

   3.2 Calculate the ith component of $\frac{\partial w}{\partial \lambda} \cdot \nabla_w \hat{y} = \nabla_\lambda \hat{y}$, the product of the Jacobian and the gradient with respect to w. The ith component is given by $(\nabla_\lambda \hat{y})_i = \sum_{j=1}^{k} (\nabla_w \hat{y})_j \frac{\partial w_j}{\partial \lambda_i}$.
   $\sigma := 0$.
   For $j = 1$ to k,

   - Calculate

     $$\frac{\partial d_j^2}{\partial \lambda_i} := 2\lambda_i (q_i - x_{j,i})^2.$$

   - Calculate

     $$\frac{\partial w_j}{\partial \lambda_i} := \frac{d\,\mathrm{W}(\rho_j)}{d\rho_j} \left(\frac{\partial d_j^2}{\partial \lambda_i} d_{k+1}^2 - d_j^2 \frac{\partial d_{k+1}^2}{\partial \lambda_i} \right) \left(\frac{1}{d_{k+1}^2} \right)^2.$$

   - Add term to sum,

     $$\sigma := \sigma + (\nabla_w \hat{y})_j \frac{\partial w_j}{\partial \lambda_i}.$$

   - Next j.

   3.3 Assign sum σ to ith element of the gradient, $(\nabla_\lambda \hat{y})_i = \frac{\partial \hat{y}}{\partial \lambda_i}$:

   $$\frac{\partial \hat{y}}{\partial \lambda_i} := \sigma.$$

   3.4 Next i.

A.4 Input Gradient

This section derives the gradient of the model output with respect to the model inputs, $\nabla_q \hat{y}$ where $q \in \mathbb{R}^{n_d}$. The jth element of this gradient is given by

$$\frac{\partial \hat{y}}{\partial q_j} = \frac{\partial}{\partial q_j} \left(\tilde{q}^{\mathrm{T}} \nu \right),$$

where $\tilde{q}^{\mathrm{T}} \triangleq [q^{\mathrm{T}} \ 1]$, and

$$\begin{aligned}
\frac{\partial \hat{y}}{\partial q_j} &= \frac{\partial}{\partial q_j} \left(\sum_{i=1}^{n_d+1} \nu_i \tilde{q}_i \right), \\
&= \sum_{i=1}^{n_d+1} \nu_j \frac{\partial \tilde{q}_i}{\partial q_j} + \frac{\partial \nu_i}{\partial q_j} \tilde{q}_i, \\
&= \nu_j + \sum_{i=1}^{n_d+1} \frac{\partial \nu_i}{\partial q_j} \tilde{q}_i,
\end{aligned}$$

where ν is given by Equation A.1. Since the linear coefficients only depend on q_j through the penalty weights, w, the chain rule can be used to write the partial derivative as

$$\begin{aligned}
\frac{\partial \hat{y}}{\partial q_j} &= \nu_j + \sum_{\ell=1}^{k} \sum_{i=1}^{n_d+1} \frac{\partial \nu_i}{\partial w_\ell} \frac{\partial w_\ell}{\partial q_j} \tilde{q}_i, \\
&= \nu_j + \sum_{\ell=1}^{k} \frac{\partial w_\ell}{\partial q_j} \left(\sum_{i=1}^{n_d+1} \frac{\partial \nu_i}{\partial w_\ell} \tilde{q}_i \right), \\
&= \nu_j + \sum_{\ell=1}^{k} \frac{\partial w_\ell}{\partial q_j} \frac{\partial}{\partial w_\ell} \left(\sum_{i=1}^{n_d+1} \nu_i \tilde{q}_i \right), \\
&= \nu_j + \sum_{\ell=1}^{k} \frac{\partial w_\ell}{\partial q_j} \frac{\partial \hat{y}}{\partial w_\ell}, \\
&= \nu_j + \frac{\partial w^{\mathrm{T}}}{\partial q_j} \nabla_w \hat{y}.
\end{aligned}$$

Each element of the input gradient can be combined into a vector to give

$$\nabla_q \hat{y} = \nu + \frac{\partial w}{\partial q} \nabla_w \hat{y},$$

where $\frac{\partial w}{\partial q}$ is the Jacobian of w with respect to q and $\nabla_w \hat{y}$ is the penalty weight gradient given by Equation A.6.

The next section describes how to calculate the Jacobian and Section A.4.2 contains an algorithm that describes how to calculate the input gradient.

A.4.1 Input Jacobian

This section derives a solution for the Jacobian of the penalty weight vector with respect to the model inputs, $\frac{\partial w}{\partial q} \in \mathbb{R}^{k \times n_d}$. This derivation is very similar to that of the metric Jacobian given in Section A.3.2.

The weights depend on the inputs indirectly through a local weighting function,

$$w_i \triangleq \mathrm{W}(\rho_i),$$

where $\rho_i \triangleq \frac{d_i^2}{d_{k+1}^2}$ and d_i is the weighted Euclidean distance between the ith nearest neighbor and the query point,

$$\begin{aligned} d_i^2 &\triangleq \mathrm{D}_{\mathrm{WE}}(q, x_i)^2, \\ &= \sum_{\ell=1}^{n_d} \lambda_\ell^2 \left(q_\ell - x_{i,\ell}\right)^2. \end{aligned}$$

The Jacobian's value in the jth row and the ith column can be obtained by following the same steps as in Equation A.7 to give

$$\frac{\partial w_i}{\partial q_j} = \frac{d\,\mathrm{W}(\rho_i)}{d\rho_i} \left(\frac{\partial d_i^2}{\partial q_j} d_{k+1}^2 - d_i^2 \frac{\partial d_{k+1}^2}{\partial q_j} \right) \left(\frac{1}{d_{k+1}^2} \right)^2.$$

The remaining unknown derivatives of this equation are readily obtained from the definition of the distance metric,

$$\begin{aligned} \frac{\partial d_i^2}{\partial q_j} &= \frac{\partial}{\partial q_j} \left(\sum_{\ell=1}^{n_d} \lambda_\ell^2 \left(q_\ell - x_{i,\ell}\right)^2 \right), \\ &= 2\lambda_j^2 (q_j - x_{i,j}). \end{aligned}$$

Once the Jacobian is calculated the gradient $\nabla_q \hat{y}$ can be calculated by applying Equation A.4. The next section summarizes these steps in an algorithm.

A.4.2 Input Gradient Algorithm

The process for calculating the gradient $\nabla_q \hat{y}$ is summarized below. This algorithm assumes that the $k+1$ nearest neighbors have been found and that the local model has been constructed as described in Section A.1. Note that the first three steps are identical to the algorithm for calculating the metric gradient, $\nabla_\lambda \hat{y}$.

Algorithm A.2: Calculation of $\nabla_q \hat{y}$

1. Calculate the penalty weight gradient:

   $\nabla_w \hat{y} = \operatorname{diag}(AS^{-1}\tilde{q}) \operatorname{diag}(b - A\nu) 2w.$

2. Calculate the derivative of the biweight function,

   $\dfrac{d\,\mathrm{W}(\rho_i)}{d\rho_i} = -\rho,$

   for each i, where

   $\rho_i \triangleq \dfrac{d_i^2}{d_{k+1}^2},$

   and d_i is the weighted Euclidean distance to the ith nearest neighbor.

3. Calculate the product of the Jacobian and $\nabla_w \hat{y}$.
   For $i = 1$ to n_d,

   3.1 Calculate

   $\dfrac{\partial d_{k+1}^2}{\partial q_i} := 2\lambda_i^2 (q_i - x_{k+1,i}).$

   3.2 Calculate the ith component of $\frac{\partial w}{\partial q} \cdot \nabla_w \hat{y} = \nabla_q \hat{y}$, the product of the Jacobian and the gradient with respect to w. The ith component is given by $(\nabla_q \hat{y})_i = \nu_i + \sum_{j=1}^{k} (\nabla_w \hat{y})_j \frac{\partial w_j}{\partial q_i}$.
   $\sigma := \nu_i$.
   For $j = 1$ to k,

   - Calculate

     $\dfrac{\partial d_j^2}{\partial q_i} := 2\lambda_i^2 (q_i - x_{j,i}).$

   - Calculate

     $\dfrac{\partial w_j}{\partial q_i} := \dfrac{d\,\mathrm{W}(\rho_j)}{d\rho_j} \left(\dfrac{\partial d_j^2}{\partial q_i} d_{k+1}^2 - \dfrac{\partial d_{k+1}^2}{\partial q_i} d_j^2 \right) \left(\dfrac{1}{d_{k+1}^2} \right)^2.$

   - Add term to sum,

     $\sigma := \sigma + (\nabla_w \hat{y})_j \dfrac{\partial w_j}{\partial q_i}.$

   - Next j.

   3.3 Assign sum σ to ith element of the gradient, $(\nabla_q \hat{y})_i = \frac{\partial \hat{y}}{\partial q_i}$:

   $\dfrac{\partial \hat{y}}{\partial q_i} := \sigma.$

3.4 Next i.

A.5 Ridge Gradient

This section derives the gradient of the model output with respect to the ridge parameters, $\nabla_r \hat{y} \in \mathbb{R}^{n_d}$ where $r \in \mathbb{R}^{n_d}$. The model output only depends on the ridge parameters through the matrix S,

$$\hat{y} \;=\; \tilde{q}^{\mathrm{T}} S^{-1} A^{\mathrm{T}} W^2 b,$$

where $S \triangleq (A^T W^2 A + R^2)$. The ith component of the gradient is given by

$$\begin{aligned}\frac{\partial y}{\partial r_i} &= \frac{\partial}{\partial r_i}\left(\tilde{q}^{\mathrm{T}} S^{-1} A^{\mathrm{T}} W^2 b\right), \\ &= \tilde{q}^{\mathrm{T}} \left(\frac{\partial}{\partial r_i} S^{-1}\right) A^{\mathrm{T}} W^2 b,\end{aligned}$$

and using Equation A.5 we have,

$$\begin{aligned}&= -\tilde{q}^{\mathrm{T}} S^{-1} \left(\frac{\partial}{\partial r_i} S\right) S^{-1} A^{\mathrm{T}} W^2 b, \\ &= -\tilde{q}^{\mathrm{T}} S^{-1} \left(\frac{\partial}{\partial r_i}\left(A^T W^2 A + R^2\right)\right) \nu, \\ &= -\tilde{q}^{\mathrm{T}} S^{-1} \left(e_i 2 r_i e_i^{\mathrm{T}}\right) \nu, \\ &= (-2 r_i)\left(\tilde{q}^{\mathrm{T}} S^{-1} e_i\right)\left(e_i^{\mathrm{T}} \nu\right), \\ &= (-2 r_i)\left(e_i^{\mathrm{T}} S^{-1} q^{\mathrm{T}}\right)\left(e_i^{\mathrm{T}} \nu\right),\end{aligned}$$

where the parentheses show that the equation reduces to a product of three scalars. Each element of the gradient can be combined into a vector to give the final expression for the ridge gradient,

$$\nabla_r \hat{y} \;=\; -2\,\mathrm{diag}(S^{-1} q)\,\mathrm{diag}(\nu)\, r. \tag{A.8}$$

Bibliography

[1] Mark Allen Weiss. *Algorithms, Data Structures, and Problem Solving with C++.* Addison-Wesley Publishing Company, Inc., 1996.

[2] Simon Haykin. *Neural Networks: A Comprehensive Foundation.* Macmillan College Publishing Company, 1994.

[3] B. Widrow, D. E. Rumelhart, and M. A. Lehr. Neural networks: Applications in industry, business and science. *Communications of the ACM*, 37(3):93–105, March 1994.

[4] W. E. Staib and J. N. McNames. Neural networks applied to steelmaking. In Michael A Arbib, editor, *The Handbook of Brain Theory and Neural Networks*, pages 934–936. MIT Press, 1995.

[5] S. H. Chen and J. S. Pan. Fast search algorithm for VQ-based recognition of isolated words. *IEE Proceedings I (Communications, Speech and Vision)*, 136(6):391–396, December 1989.

[6] Tung-Shou Chen and Chin-Chen Chang. Diagonal axes method (DAM): A fast search algorithm for vector quantization. *IEEE Transactions on Circuits and Systems for Video Technology*, 7(3):555–559, June 1997.

[7] Christopher G. Atkeson, Andrew W. Moore, and Stefan Schaal. Locally weighted learning. *Artificial Intelligence Review*, 11(1–5):11–73, February 1997.

[8] Thomas Schreiber. Efficient neighbor searching in nonlinear time series analysis. *International Journal of Bifurcation and Chaos*, 5(2):349–358, 1995.

[9] James McNames. A nearest trajectory strategy for time series prediction. In *Proceedings of the International Workshop on Advanced Black-Box Techniques for Nonlinear Modeling*, pages 112–128, Katholieke Universiteit Leuven, Belgium, July 1998.

[10] C. M. Eastman and S. F. Weiss. Tree-structures for high dimensionality neighbor searching. *Information Systems*, 7(2):115–122, 1982.

[11] C.Y. Chen, C. C. Chang, and R. C. T. Lee. A near pattern-matching scheme based upon principal component analysis. *Pattern Recognition Letters*, 16:339–345, April 1995.

[12] Stelios G. Bakamidis. An exact fast nearest neighbor identification technique. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 5, pages 658–661, 1993.

[13] De-Yuan Cheng, Allen Gersho, Bhaskar Ramamurthi, and Yair Shoham. Fast search algorithms for vector quantization and pattern matching. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 1, pages 9.11.1–9.11.4, March 1984.

[14] J. Jiménez, J. A. Moreno, and G. J. Ruggeri. Forecasting on chaotic time series: A local optimal linear-reconstruction method. *Physical Review A*, 45(6):3553–3558, March 1992.

[15] H. Waelbroeck, R. López-Peña, T. Morales, and F. Zertuche. Prediction of tropical rainfall by local phase space reconstruction. *Journal of the Atmospheric Sciences*, 51(22):3360–3364, 1994.

[16] Guillermo J. Berri and Jan Paegle. Sensitivity of local predictions to initial conditions. *Journal of Applied Meteorology*, 29:256–267, 1990.

[17] J. B. Elsner and A. A. Tsonis. Nonlinear prediction, chaos, and noise. *Bulletin American Meteorological Society*, 73(1):49–60, January 1992.

[18] Martin Casdagli, Deirdre Des Jardins, Stephen Eubank, J. Doyne Farmer, John Gibson, and James Theiler. Nonlinear modeling of chaotic time series: Theory and applications. In Jong Hyun Kim and John Stringer, editors, *Applied Chaos*, pages 335–380. John Wiley & Sons, Inc., 1992.

[19] George Sugihara and Robert M. May. Nonlinear forecasting as a way of distinguishing chaos from measurement error in time series. *Nature*, 344:734–741, April 1990.

[20] George Sugihara. Nonlinear forecasting for the classification of natural time series. *Philosophical Transactions of the Royal Society of London A*, 348:477–495, 1994.

[21] E. A. Wan. Combining fossil and sunspot data: Committee predictions. In *1997 IEEE International Conference on Neural Networks*, pages 2176–80, June 1997.

[22] Martin Casdagli. Chaos and determinisitc versus stochastic non-linear modelling. *J. R. Statist. Soc. B*, 54(2):303–28, 1992.

[23] Michael D. Mundt, W. Bruce Maguire, and Robert R. P. Chase. Chaos in the sunspot cycle: Analysis and prediction. *Journal of Geophysical Research*, 96(A2):1705–1716, 1989.

[24] Kevin Judd and Alistair Mees. On selecting models for nonlinear time series. *Physica D*, 82:426–444, 1995.

[25] H. D. Navone and H. A. Ceccatto. Forecasting chaos from small data sets: a comparison of different nonlinear algorithms. *Journal of Physics A: Mathematics*, 28:3381–3388, 1995.

[26] P. García, J. Jiménez, A. Marcano, and F. Moleiro. Local optimal metrics and nonlinear modeling of chaotic time series. *Physical Review Letters*, 76(9):1449–1452, February 1996.

[27] D. Kugiumtzis. State space reconstruction parameters in the analysis of chaotic time series — the role of the time window length. *Physica D*, 95:13–28, 1996.

[28] Andreas S. Weigend and Neil A. Gershenfeld. *Time Series Prediction*. Addison-Wesley Publishing Company, 1994.

[29] Henry D. I. Abarbanel, Z. Gills, C. Liu, and R. Roy. Nonlinear-time-series analysis of chaotic laser dynamics. *Physical Review A*, 53(1):440–453, January 1996.

[30] D. S. Broomhead, J. P. Huke, and Potts M. A. S. Cancelling deterministic noise by constructing nonlinear inverses to linear filters. *Physica D*, 89:439–458, 1996.

[31] Santishmohan T. S. Bukkapatnam, Akhlesh Lakhtakia, and Soundar R. T. Kumara. Analysis of sensor signals shows turning on a lathe exhibits low-dimensional chaos. *Physical Review E*, 52(3):2375–2387, September 1995.

[32] Lebender D. and F. W. Schneider. Neural nets and the local predictor method used to predict the time series of chemical reactions. *The Journal of Physical Chemistry*, 97(34):8764–8769, 1993.

[33] Jerome H. Friedman. An overview of predictive learning and function approximation. In Vladimir Cherkassky, Jerome H. Friedman, and Harry Wechsler, editors, *From Statistics to Neural Networks*, volume 136 of *Computer and Systems Sciences*, pages 1–61. Springer-Verlag, 1994.

[34] J. D. Farmer and John J. Sidorowich. Exploiting chaos to predict the future and reduce noise. In Yee Chung Lee, editor, *Evolution, Learning and Cognition*, pages 277–330. World Scientific, 1988.

[35] J. Doyne Farmer and John J. Sidorowich. Predicting chaotic time series. *Physical Review Letters*, 59(8):845–848, August 1987.

[36] Zhong Liu, Xiaolin Ren, and Zhiwen Zhu. Equivalence between different local prediction methods of chaotic time series. *Physics Letters A*, 227:37–40, March 1997.

[37] Tim Sauer. Time series prediction by using delay coordinate embedding. In Andreas S. Weigend and Neil A. Gershenfeld, editors, *Time Series Prediction*, Santa Fe Institue Studies in the Sciences of Complexity, pages 175–193. Addison-Wesley, 1994.

[38] Eric A. Wan. Time series prediction by using a connectionist network with internal delay lines. In Andreas S. Weigend and Neil A. Gershenfeld, editors, *Time Series Prediction*, Santa Fe Institue Studies in the Sciences of Complexity, pages 195–217. Addison-Wesley, 1994.

[39] Johan A. K. Suykens and Joos Vandewalle. *Nonlinear Modeling Advanced Black-Box Techniques*. Kluwer Academic Publishers, 1998.

[40] J.A.K. Suykens and J. Vandewalle, editors. *Proceedings of the International Workshop on Advanced Black-Box Techniques for Nonlinear Modelings*, Katholieke Universiteit Leuven, Belgium, July 1998.

[41] Jeffrey S. Simonoff. *Smoothing Methods in Statistics*. Statistics. Springer-Verlag, 1996.

[42] Gilbert Strang. *Linear Algebra and Its Applications*. Harcourt Brace Jovanovich College Publishers, third edition, 1988.

[43] Mokhtar S. Bazaraa, Hanif D. Sherali, and C. M. Shetty. *Nonlinear Programming: Theory and Algorithms.* John Wiley & Sons, Inc., second edition, 1993.

[44] D. Kugiumtzis, N. Lingjærde, and N. Christophersen. Regularized local linear prediction of chaotic time series. *Physica D*, 112:344–360, 1998.

[45] D. Kugiumtzis Lillekjendlie and N. Christophersen. Chaotic time series. Part II. System identification and prediction. *Modeling, Identification and Control*, 15(4):225–243, October 1994.

[46] Peter Lancaster and Kęstutis Šalkauskas. *Curve and Surface Fitting: An Introduction.* Academic Press Inc., 1986.

[47] Vladimir Naumovich Vapnik. *The nature of statistical learning theory.* Springer-Verlag, 1995.

[48] Martin C. Casdagli and Andreas S. Weigend. Exploring the continuum between deterministic and stochastic modeling. In Andreas S. Weigend and Neil A. Gershenfeld, editors, *Time Series Prediction*, Santa Fe Institute Studies in the Sciences of Complexity, pages 347–366. Addison-Wesley, 1994.

[49] András Faragó, Tamás Linder, and Gábor Lugosi. Faster nearest-neighbor search in dissimilarity spaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):957–962, September 1993.

[50] Enrique Vidal. New formulation and improvements of the nearest-neighbor approximating and eliminating search algorithm (AESA). *Pattern Recognition Letters*, 15:1–7, January 1994.

[51] Sanjiv Kapoor and Michiel Smid. New techniques for exact and approximate dynamic closest-point problems. *SIAM Journal on Computing*, 25(4):775–796, August 1996.

[52] V. Ramasubramanian and Kuldip K. Paliwal. Fast k-dimensional tree algorithms for nearest neighbor search with application to vector quantization encoding. *IEEE Transactions on Signal Processing*, 40(3):518–531, March 1992.

[53] C.-Q. Chen, S.-N. Koh, and I.-Y. Soon. Fast codebook search algorithm for unconstrained vector quantisation. *IEE Proceedings- Vision, Image, and Signal Processing*, 145(2):97–102, April 1998.

[54] K.-T. Lo and W.-K. Cham. Subcodebook searching algorithm for efficient VQ encoding of images. *IEE Proceedings-I*, 140(5):327–330, October 1993.

[55] Chang-Hsing Lee and Ling-Hwei Chen. A fast search algorithm for vector quantization using mean pyramids of codewords. *IEEE Transactions on Communications*, 43(2–4):1697–1702, 1995.

[56] N. Santosh and C. Eswaran. Efficient search algorithm for fast encoding of images using vector quantisation. *Electronics Letters*, 32(23):2135–2137, November 1996.

[57] Wen-Jyi Hwang and Biing-Yau Chen. Fast vector quantisation encoding algorithm using zero-tree data structure. *Electronics Letters*, 33(15):1290–1292, 1997.

[58] I. K. Sethi. A fast algorithm for recognizing nearest neighbors. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-11(3):245–248, 1981.

[59] De-Yuan Cheng and Allen Gersho. A fast codebook search algorithm for nearest-neighbor pattern matching. In *IEEE-IECEJ-ASJ International Conference on Acoustics, Speech and Signal Processing*, volume 1, pages 265–268, April 1986.

[60] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582, January 1994.

[61] Sameer A. Nene and Shree K. Nayar. Closest point search in high dimensions. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 859–865, June 1996.

[62] Sameer A. Nene and Shree K. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(9):989–1003, September 1997.

[63] Jim Z. C. Lai. Fast encoding algorithms for tree-structured vector quantization. *Image and Vision Computing*, 15:867–871, 1997.

[64] Sunil Arya, David M. Mount, Ruth Netanyahu, Nathan S. Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, 45(6):891–923, November 1998.

[65] Mohammad Reza Soleymani and Salvatore D. Morgera. An efficient nearest neighbor search method. *IEEE Transactions on Communications*, 35(6):677–679, June 1987.

[66] G. Poggi. Fast algorithm for full-search VQ encoding. *Electronics Letters*, 29(12):1141–1142, June 1993.

[67] J. S. Pan, F. R. McInnes, and M. A. Jack. Bound for minkowski metric or quadratic metric applied to VQ codeword search. *IEE Proceedings-Vision, Image and Signal Processing*, 143(1):67–71, 1996.

[68] L. Guan and M. Kamel. Equal-average hyperplane partitioning method for vector quantization of image data. *Pattern Recognition Letters*, 13:693–699, 1992.

[69] S.W. Ra and J. K. Kim. A fast mean-distance-ordered partial codebook search algorithm for image vector quantization. *IEEE Transactions on Circuits and Systems*, 40(9):576–579, September 1993.

[70] Chang-Hsing Lee and Ling-Hwei Chen. High-speed closest codeword search algorithms for vector quantization. *Signal Processing*, 43:323–331, 1995.

[71] C.-H. Lee and L.-H. Chen. Fast closest codeword search algorthm[sic] for vector quantisation. *IEE proceedings. Vision, image and signal processing*, 141(3):143–148, June 1994.

[72] SeongJoon Baek, BumKi Jeon, and Koeng-Mo Sung. A fast encoding algorithm for vector quantization. *IEEE Signal Processing Letters*, 4(12):325–327, December 1997.

[73] L. Torres and J. Huguet. An improvement on codebook search for vector quantization. *IEEE Transactions on Communications*, 42(2–4):208–210, April 1994.

[74] Jack Bryant. A fast classifier for image data. *Pattern Recognition*, 22(1):45–48, 1989.

[75] K. K. Paliwal and V. Ramasubramanian. Effect of ordering the codebook on the efficiency of the partial distance search algorithm for vector quantization. *IEEE Transactions on Communications*, 37(5):538–540, May 1989.

[76] Yuk-Hee Chan and Siu Wan-chi. In search of the optimal searching sequence for VQ encoding. *IEEE Transactions on Communications*, 43(12):2891–2893, December 1995.

[77] Patrick J. Grother, Gerald T. Candela, and James L. Blue. Fast implementations of nearest neighbor classifiers. *Pattern Recognition*, 30(3):459–465, 1997.

[78] Jerome H. Friedman, Forest Baskett, and Leonard J. Shustek. An algorithm for finding nearest neighbors. *IEEE Transactions on Computers*, C-24:1000–1010, October 1975.

[79] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[80] Baek S. Kim and Song B. Park. A fast k nearest neighbor finding algorithm based on the ordered partition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):761–766, November 1986.

[81] Keinosuke Fukunaga and Patrenahalli M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Transactions on Computers*, C-24:750–753, July 1975.

[82] Luisa Micó, José Oncina, and Rafael C. Carrasco. A fast branch & bound nearest neighbor classifier in metric spaces. *Pattern Recognition Letters*, 17:731–739, 1996.

[83] Ioannis Katsavounidis, C.-C. Jay Kuo, and Zhen Zhang. Fast tree-structured nearest neighbor encoding for vector quantization. *IEEE Transactions on Image Processing*, 5(2):398–404, February 1996.

[84] Thomas P. Yunck. A technique to identify nearest neighbors. *IEEE Transactions on Systems, Man, and Cybernetics*, 6(10):678–683, October 1976.

[85] H. Niemann and R. Goppert. An efficient branch-and-bound nearest neighbor classifier. *Pattern Recognition Letters*, 7:67–72, February 1988.

[86] Qiyuan Jiang and Wenshu Zhang. An improved method for finding nearest neighbors. *Pattern Recognition Letters*, 14:531–535, July 1993.

[87] Abdelhamid Djouadi and Essaid Bouktache. A fast algorithm for the nearest-neighbor classifier. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(3):277–282, March 1997.

[88] Stephane Lubiarz and Philip Lockwood. Evaluation of fast algorithms for finding the nearest neighbor. In *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 1491–1494, April 1997.

[89] Pierre Zakarauskas and John M. Ozard. Complexity analysis for partitioning nearest neighbor searching algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(6):663–668, June 1996.

[90] B. G. Batchelor. Faster nearest-neighbor calculations. *Electronics Letters*, 13(10):304–306, May 1977.

[91] Iraj Kalantari and Gerard McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Transactions on Software Engineering*, SE-9(5):631–634, September 1983.

[92] Behrooz Kamgar-Parsi and Laveen N. Kanal. An improved branch and bound algorithm for computing k-nearest neighbors. *Pattern Recognition Letters*, 3:7–12, January 1985.

[93] Philip Lockwood. A low cost DTW-based discrete utterance recogniser. In *Eighth International Conference on Pattern Recognition*, pages 467–469, 1986.

[94] Enrique Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, July 1986.

[95] V. Ramasubramanian and K. K. Paliwal. An efficient approximation-elimination algorithm for fast nearest neighbor search based on a spherical distance coordinate formulation. In *Signal Processing V: Theories and Applications*, volume 2, pages 1323–1326, 1990.

[96] Michael T. Orchard. A fast nearest-neighbor search algorithm. In *1991 International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 2297–2300, April 1991.

[97] S. O. Belkasim, M. Shridhar, and M. Ahmadi. Pattern classification using and efficient KNNR. *Pattern Recognition*, 25(10):1269–1274, 1992.

[98] C.-M. Huang, Q. Bi, G. S. Stiles, and R. W. Harris. Fast full search equivalent encoding algorithms for image compression using vector quantization. *IEEE Transactions on Image Processing*, 1(3):413–416, July 1992.

[99] Luisa Micó, José Oncina, and Enrique Vidal. An algorithm for finding nearest neighbors in constant average time with linear space complexity. In *Proceedings of the*

11th International Conference on Pattern Recognition, volume II, pages 557–560, The Hague, 1992.

[100] E. Salari and W. Li. Adaptive fast encoding algorithm for vector quantisation. *Electronics Letters*, 30(21):1733–1734, October 1994.

[101] María Luisa Micó, José Oncina, and Enrique Vidal. A new version of the nearest-neighbor approximating and eliminating search algorithm (AESA) with linear preprocessing time and memory requirements. *Pattern Recognition Letters*, 15:9–17, January 1994.

[102] Wenhua Li and Ezzatollah Salari. A fast vector quantization encoding method for image compression. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(2):119–123, April 1995.

[103] Kuang-Shyr Wu and Ja-Chen Lin. An efficient nearest neighbor searching algorithm with application to LBG codebook generation. *Journal of the Chinese Institute of Engineers*, 19(6):719–724, 1996.

[104] S. C. Tai, C. C. Lai, and Y. C. Lin. Two fast nearest neighbor searching algorithms for image vector quantization. *IEEE Transactions on Communications*, 44(12):1623–1628, December 1996.

[105] Yih-Chuan Lin and Shen-Chuan Tai. Dynamic windowed codebook search algorithm in vector quantization. *Optical Engineering*, 35(10):2921–2929, October 1996.

[106] Chin-Chen Chang and Dai-Chuan Lin. An improved VQ codebook search algorithm using principal component analysis. *Journal of Visual Communication and Image Representation*, 8(1):27–37, 1997.

[107] Chin-Chen Chang, Wen-Tsai Li, and Tung-Shou Chen. Two improved codebook search methods of vector quantization based on orthogonal checking and fixed range search. *Journal of Electronic Imaging*, 7(2):357–366, April 1998.

[108] Keinosuke Fukunaga. *Introduction to Statistical Pattern Recognition*. Academic Press, Inc., 1990.

[109] Karl W. Pettis, Thomas A. Bailey, Anil K. Jain, and Richard C. Dubes. An intrinsic dimensionality estimator from near-neighbor information. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1(1):25–37, 1979.

[110] Chang-Da Bei and Robert M. Gray. An improvement of the minimum distortion encoding algorithm for vector quantization. *IEEE Transactions on Communications*, 33(10):1132–1133, October 1985.

[111] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[112] Pasi Fränti, Timo Kaukoranta, and Olli Nevalainen. On the splitting method for vector quantization codebook generation. *Optical Engineering*, 36(11):3043–3051, November 1997.

[113] Edward W. Forgy. Cluster analysis of multivariate data: Efficiency vs. interpretibility of classification. *Biometrics*, 21:768–769, 1965. Abstract.

[114] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematics, Statistics, and Probability*, volume 1, pages 281–296, 1967.

[115] Yoseph Linde, Andrés Buzo, and Robert M. Gray. An algorithm for vector quantizer design. *IEEE Transactions on Communications*, 28(1):84–95, January 1980.

[116] G. H. Golub, M. Heath, and G. Wahba. Generalized cross-validation as a method for choosing a good ridge parameter. *Technometrics*, 21:215–223, 1979.

[117] John Neter, Michael H. Kutner, Christopher J. Nachtsheim, and William Wasserman. *Applied Linear Statistical Models*. Richard D. Irwin, Inc., fourth edition, 1996.

[118] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1992.

[119] David G. Luenberger. *Linear and Nonlinear Programming*. Addison-Wesley, second edition, 1984.

[120] Daniel B. Murray. Forecasting a chaotic time series using an improved metric for embedding space. *Physica D*, 68:318–325, 1993.

[121] Naoki Tanaka, Hiroshi Okamoto, and Masayoshi Naito. An optimal metric for predicting chaotic time series. *Japanese Journal of Applied Physics*, 34(1):388–394, January 1995.

[122] Klaus Fraedrich and Bernd Rückert. Metric adaption for analog forecasting. *Physica A*, 253:379–393, 1998.

[123] Bernard Widrow and Samuel D. Stearns. *Adaptive Signal Processing*. Prentice-Hall, Inc., 1985.

[124] Hassan K. Khalil. *Nonlinear Systems*. Prentice Hall, second edition, 1996.

[125] Martin Casdagli, Stephen Eubank, J. Doyne Farmer, and John Gibson. State space reconstruction in the presence of noise. *Physica D*, 51:52–98, 1991.

[126] Rainer Hegger and Thomas Schreiber. A noise reduction method for multivariate time series. *Physics Letters A*, 170(4):305–310, 1992.

[127] Tim Sauer. A noise reduction method for signals from nonlinear systems. *Physica D*, 58:193–201, 1992.

[128] Thomas Schreiber. Extremely simple nonlinear noise-reduction method. *Physical Review E*, 47(4):2401–2404, April 1993.

[129] Mike Davies. Noise reduction schemes for chaotic time series. *Physica D*, 79:174–192, 1994.

[130] Holger Kantz and Lars Jaeger. Improved cost functions for modelling of noisy chaotic time series. *Physica D*, 109:59–69, 1997.

[131] Lars Jaeger and Holger Kantz. Effective determinisitic models for chaotic dynamics perturbed by noise. *Physical Review E*, 55(5):5234–5247, May 1997.

[132] F. Takens. Detecting strange attractors in turbulence. In D. A. Rand and L. S. Young, editors, *Dynamical Systems and Turbulence*, volume 898 of *Lecture Notes in Mathematics*, pages 336–381. Springer-Verlag, 1981.

[133] Tim Sauer, James A. Yorke, and Martin Casdagli. Embedology. *Journal of Statistical Physics*, 65(3):579–616, 1991.

[134] Th. Buzug and G. Pfister. Comparison of algorithms calculating optimal embedding parameters for delay time coordinates. *Physica D*, 58:127–137, 1992.

[135] Liangyue Cao. Practical method for determining the minimum embedding dimension of a scalar time series. *Physica D*, 110:43–50, 1997.

[136] Martin Casdagli. Nonlinear prediction of chaotic time series. *Physica D*, 35:335–356, 1989.

[137] D. Kugiumtzis. Assessing different norms in nonlinear analysis of noisy time series. *Physica D*, 105:62–78, 1997.

[138] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–80, 1989.

[139] P. J. Werbos. Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–60, October 1990.

[140] Katheleen T. Alligood, Tim D. Sauer, and James A Yorke. *Chaos, An Introduction to Dynamical Systems.* Textbooks in Mathematical Sciences. Springer-Verlag New York, Inc., 1997.

[141] Udo Hübner, Carl-Otto Weiss, Neal Broadus Abraham, and Dingyuan Tang. Lorenz-like chaos in NH_3-FIR lasers. In Andreas S. Weigend and Neil A. Gershenfeld, editors, *Time Series Prediction*, Santa Fe Institue Studies in the Sciences of Complexity, pages 73–104. Addison-Wesley, 1994.

[142] T. L. Carroll. Multiple attractors and periodic transients in synchronized nonlinear circuits. *Physics Letters A*, 238(6):365–368, 1998.

[143] E. R. Weeks, Y. Tian, J. S. Urbach, K. Ide, H. L. Swinney, and M. Ghil. Transitions between blocked and zonal flows in a rotating annulus with topography. *Science*, 278:1598–1601, 1997.

[144] Leonard A. Smith. Local optimal prediction: exploiting strangeness and the variation of sensitivity to initial condition. In *Philosophical Transactions of the Royal Society*, volume 348 of *A*, pages 371–381, 1994.

[145] G. Bontempi, M. Birattari, and H. Bersini. Lazy learning for iterated time-series prediction. In *Proceedings of the International Workshop on Advanced Black-Box Techniques for Nonlinear Modeling*, pages 62–68, Katholieke Universiteit Leuven, Belgium, July 1998.

[146] Katsuhiko Ogata. *Modern Control Engineering.* Prentice Hall, second edition, 1990.