GREEDY DICTIONARY LEARNING ALGORITHMS FOR SPARSE SURROGATE MODELLING

by

Valentin Stolbunov

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of the Institute for Aerospace Studies
University of Toronto

© Copyright 2017 by Valentin Stolbunov
Abstract
Greedy Dictionary Learning Algorithms for Sparse Surrogate Modelling

Valentin Stolbunov
Doctor of Philosophy
Graduate Department of the Institute for Aerospace Studies
University of Toronto
2017

In the field of engineering design, numerical simulations are commonly used to forecast system performance before physical prototypes are built and tested. However the fidelity of predictive models has outpaced advances in computer hardware and numerical methods, making it impractical to directly apply numerical optimization algorithms to the design of complex engineering systems modelled with high fidelity. A promising approach for dealing with this computational challenge is the use of surrogate models, which serve as approximations of the high-fidelity computational models and can be evaluated very cheaply. This makes surrogates extremely valuable in design optimization and a wider class of problems: inverse parameter estimation, machine learning, uncertainty quantification, and visualization.

This thesis is concerned with the development of greedy dictionary learning algorithms for efficiently constructing sparse surrogate models using a set of scattered observational data. The central idea is to define a dictionary of basis functions either \textit{a priori} or \textit{a posteriori} in light of the dataset and select a subset of the basis functions from the dictionary using a greedy search criterion. In this thesis, we first develop a novel algorithm for sparse learning from parameterized dictionaries in the context of greedy radial basis function learning (GRBF). Next, we develop a novel algorithm for general dictionary learning (GGDL). This algorithm is presented in the context of multiple kernel learning with heterogenous dictionaries. In addition, we present a novel strategy, based on cross-validation, for parallelizing greedy dictionary learning and a randomized sampling strategy to significantly reduce approximation costs associated with large dictionaries.

We also employ our GGDL algorithm in the context of uncertainty quantification to construct sparse polynomial chaos expansions. Finally, we demonstrate how our algorithms may be adapted to approximate gradient-enhanced datasets.

Numerical studies are presented for a variety of test functions, machine learning datasets, and engineering case studies over a wide range of dataset size and dimensionality. Compared to state-of-the-art approximation techniques such as classical radial basis function approximations, Gaussian process models, and support vector machines, our algorithms build surrogates which are significantly more sparse, of comparable or improved accuracy, and often offer reduced computational and memory costs.
Acknowledgements

I would like to begin by thanking Professor Prasanth Nair for being the absolute best mentor one could hope to have had. I am grateful for his support, guidance, patience, insight, and enthusiasm. Professor Nair puts his students first and it is thanks to him that I had the confidence and self-discipline to accomplish everything that I have.

I would also like to thank the other members of my doctoral examination committee, Professor Clinton Groth and Professor David Zingg, for their comments on, and critiques of, the work presented here.

To my UTIAS group colleagues, past and present – Mina Mitry, Matthew Li, Tomas Mawyin, Dr. Christophe Audouze, Ricardo Santos-Baptista, and Trefor Evans – I am honoured to have had the opportunity to collaborate with you and learn from you.

I am grateful for financial support from UTIAS, the Centre for Research in Sustainable Aviation, the Ontario Graduate Scholarship, and Pratt & Whitney Canada.

On a personal note, I would like to thank Umesh Thillaivasan, Phung Tran, and Victoria Tu. Our countless camping trips and game nights kept me sane. Over the past four years I have grown more as a person than in the previous twenty-two and for this period I am glad to have had you as my closest friends.

Finally, I would like to acknowledge, and dedicate this thesis to, my parents. It would take another 150 pages for me to thank you for everything you have done for me, so just know that I love you.
Contents

1 Introduction 1
   1.1 Motivation ................................. 1
   1.2 Thesis Objectives .......................... 3
   1.3 Thesis Outline .............................. 3

2 Background and Related Work 5
   2.1 Approximation of Scattered Data .............. 5
      2.1.1 Radial Basis Function Expansions .......... 9
      2.1.2 Kernel Methods ........................... 11
   2.2 Sparse Dictionary Learning .................. 14
      2.2.1 $\ell_1$ Minimization .................... 15
      2.2.2 Greedy Algorithms ....................... 16
   2.3 Summary and Concluding Remarks ............. 19

3 Greedy Algorithm for RBF Approximation 20
   3.1 Motivation .................................. 20
   3.2 Greedy Algorithm for RBF Approximation .... 23
      3.2.1 Greedy Selection Criteria ............... 25
      3.2.2 Model Update Strategy and Shape Parameter Tuning .. 26
      3.2.3 Choice of Stopping Criteria ............ 29
      3.2.4 Final Algorithm .......................... 31
   3.3 Tuning the RBF Kernel Type ................. 32
   3.4 Summary and Concluding Remarks ............. 33

4 Numerical Results for GRBF 35
   4.1 Demonstrative Examples ..................... 35
      4.1.1 GRBF Predictor Evolution .............. 35
      4.1.2 One-Dimensional Example ............... 38
4.1.3 Residual Convergence Trends ........................................ 42
4.2 Test Functions .................................................................. 43
  4.2.1 Case 1: \( n = 1, 2, 10n \leq m \leq 50n \) .................... 44
  4.2.2 Case 2: \( n = 10, 20, 10n \leq m \leq 50n \) ............. 45
  4.2.3 Case 3: \( n = 1, 2, m > 50n \) ............................. 46
  4.2.4 Case 4: \( n = 10, 20, m > 50n \) .......................... 48
4.3 Summary and Concluding Remarks ................................. 50

5 Greedy Algorithms for General Dictionary Learning ............ 52
  5.1 Motivation ................................................................. 52
  5.2 Multiple Kernel Learning ............................................. 53
    5.2.1 Regularized Risk Minimization with Multiple Kernels ... 54
    5.2.2 A Greedy Approach to Multiple Kernel Learning ...... 56
  5.3 Parallel Algorithm for General Dictionaries ..................... 58
    5.3.1 General Dictionaries and Problem Statement ............ 58
    5.3.2 Candidate Selection ........................................... 61
    5.3.3 Stopping Criterion: \( \nu \)-fold Cross Validation .......... 62
    5.3.4 Parallelization .................................................. 64
    5.3.5 Algorithm Block ................................................. 66
  5.4 Randomized Algorithm for Larger Dictionaries ................. 66
    5.4.1 Random Candidate Sampling ................................ 68
    5.4.2 Stopping Criteria .............................................. 70
  5.5 Summary and Concluding Remarks .................................. 70

6 Numerical Results and Case Studies for GGDL .................... 72
  6.1 Demonstrative Examples ............................................. 72
    6.1.1 GGDL-CV Predictor Evolution ............................... 73
    6.1.2 GGDL-R Predictor Evolution ................................ 76
    6.1.3 Shape Parameter Tuning .................................... 79
    6.1.4 Structurally-Sampled RBF Centres .......................... 83
    6.1.5 Learning with Multiple Kernel Types ..................... 84
  6.2 Machine Learning Datasets .......................................... 87
    6.2.1 Abalone ....................................................... 90
    6.2.2 Boston ......................................................... 91
    6.2.3 CPU Small ................................................... 91
    6.2.4 Diabetes ....................................................... 92
    6.2.5 Final Results ................................................ 93
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Common radial basis function definitions, including the Gaussian, multi-quadric (MQ), and inverse multi-quadric (IMQ).</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Common reproducing kernels, including the Gaussian, polynomial, linear, and (conditionally positive definite) sigmoid.</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>GRBF accuracy studies for the $F_1$, $F_2$, and $F_3$ test functions in $n = 1$ dimensions and both uniform and random datasets of various small sizes.</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>GRBF accuracy studies for the $F_4$, $F_5$, and $F_6$ test functions in $n = 2$ dimensions and both uniform and random datasets of various small sizes.</td>
<td>46</td>
</tr>
<tr>
<td>4.3</td>
<td>GRBF accuracy studies for the $F_7$, $F_8$, and $F_9$ test functions in $n = 10$ dimensions and Sobol-sampled datasets of various small sizes.</td>
<td>47</td>
</tr>
<tr>
<td>4.4</td>
<td>GRBF accuracy studies for the $F_7$, $F_8$, and $F_9$ test functions in $n = 20$ dimensions and Sobol-sampled datasets of various small sizes.</td>
<td>47</td>
</tr>
<tr>
<td>4.5</td>
<td>GRBF accuracy and timing studies for the $F_1$, $F_2$, and $F_3$ test functions in $n = 1$ dimensions and both uniform and random datasets of various large sizes.</td>
<td>48</td>
</tr>
<tr>
<td>4.6</td>
<td>GRBF accuracy and timing studies for the $F_4$, $F_5$, and $F_6$ test functions in $n = 2$ dimensions and both uniform and random datasets of various large sizes.</td>
<td>49</td>
</tr>
<tr>
<td>4.7</td>
<td>GRBF accuracy and timing studies for the $F_7$, $F_8$, and $F_9$ test functions in $n = 10$ dimensions and Sobol-sampled datasets of various sizes.</td>
<td>49</td>
</tr>
<tr>
<td>4.8</td>
<td>GRBF accuracy and timing studies for the $F_7$, $F_8$, and $F_9$ test functions in $n = 20$ dimensions and Sobol-sampled datasets of various sizes.</td>
<td>50</td>
</tr>
<tr>
<td>6.1</td>
<td>List of kernel types, RBF centres, and shape parameters for the final predictor from Figure 6.1.</td>
<td>75</td>
</tr>
<tr>
<td>6.2</td>
<td>List of kernel types, RBF centres, and shape parameters for the final predictors from Figure 6.3.</td>
<td>78</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>6.3</td>
<td>List of kernel types, RBF centres, and shape parameters for the final predictors from Figure 6.5</td>
<td>81</td>
</tr>
<tr>
<td>6.4</td>
<td>Average test set errors for SVM, GPM, GGDL-CV, and GGDL-R approximations over four popular machine learning datasets</td>
<td>93</td>
</tr>
<tr>
<td>6.5</td>
<td>Average training times for SVM, GPM, GGDL-CV, and GGDL-R approximations over four popular machine learning datasets</td>
<td>94</td>
</tr>
<tr>
<td>6.6</td>
<td>Average predictor sparsities for SVM, GPM, GGDL-CV, and GGDL-R approximations over four popular machine learning datasets</td>
<td>94</td>
</tr>
<tr>
<td>6.7</td>
<td>List of PC basis functions chosen at each iteration when using 4-fold GGDL-CV to approximate a Sobol-sampled dataset from the thermal driven cavity flow problem</td>
<td>99</td>
</tr>
<tr>
<td>6.8</td>
<td>Errors in the first two statistical moments of the quantity of interest from the thermal driven cavity flow problem are compared for QMC, FPC, and SPC approximations for a variety of dataset sizes and polynomial orders</td>
<td>100</td>
</tr>
<tr>
<td>7.1</td>
<td>Complexity analysis for the stages of GRBF, GGDL-CV, GGDL-R, GRBFH-P, and GRBFH-A</td>
<td>119</td>
</tr>
<tr>
<td>7.2</td>
<td>The accuracy of three RBF algorithms is compared for gradient-enhanced datasets</td>
<td>124</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Demonstration of bias-variance trade off using polynomials of orders 2, 6, and 12 to predict a random noisy sample of the function $y = \text{sinc}(x)$</td>
</tr>
<tr>
<td>2.2</td>
<td>Effect of changing the shape parameter value for three common radial basis functions</td>
</tr>
<tr>
<td>3.1</td>
<td>Comparison of RMS test error, brute force LOO, and approximate LOO for an RBF approximation of a test function</td>
</tr>
<tr>
<td>4.1</td>
<td>Evolution of a GRBF approximation for a test function</td>
</tr>
<tr>
<td>4.2</td>
<td>Training set error trend for GRBF approximation of $F_1$ test function</td>
</tr>
<tr>
<td>4.3</td>
<td>Reciprocal condition number trend for GRBF approximation of $F_1$ test function</td>
</tr>
<tr>
<td>4.4</td>
<td>Approximate LOO trend for GRBF approximation of $F_1$ test function</td>
</tr>
<tr>
<td>4.5</td>
<td>Comparison of RBF and GRBF predictors for a random sample of the $F_2$ test function</td>
</tr>
<tr>
<td>4.6</td>
<td>Training set error trend for GRBF approximation of $F_2$ test function</td>
</tr>
<tr>
<td>4.7</td>
<td>Reciprocal condition number trend for GRBF approximation of $F_2$ test function</td>
</tr>
<tr>
<td>4.8</td>
<td>Approximate LOO trend for GRBF approximation of $F_2$ test function</td>
</tr>
<tr>
<td>4.9</td>
<td>Test set error trend for GRBF approximation of $F_2$ test function</td>
</tr>
<tr>
<td>4.10</td>
<td>Comparison of two GRBF predictors of varying sparsities for a random sample of the $F_2$ test function</td>
</tr>
<tr>
<td>4.11</td>
<td>Training set error trends for a number of greedy algorithms over their approximation of the $F_4$ test function</td>
</tr>
<tr>
<td>4.12</td>
<td>Accuracy comparison between the GRBF algorithm and classical RBF algorithm for the $F_5$ test function</td>
</tr>
<tr>
<td>5.1</td>
<td>Curves of training error vs validation error for three splits of a synthetic dataset of $m = 250$ examples, and the average of the curves</td>
</tr>
</tbody>
</table>
5.2 GGDL-CV approximation of a synthetic dataset of \( m = 250 \) examples. . 65

5.3 Plots of training set error and test set error for the GGDL algorithm being applied to a 2000-example training sample from the Abalone dataset. . 69

6.1 Evolution of a GGDL-CV approximation for a test function. .......... 74

6.2 A final 9-iteration GGDL-CV predictor for a test function, with RBF centres visualized. .......................................................... 75

6.3 Evolution of a GGDL-R approximation for a test function. .......... 77

6.4 The final 9-iteration GGDL-CV predictor from Figure 6.1 is compared to the three final GGDL-R predictors from Figure 6.3. .......... 79

6.5 A noisy 50-point dataset from the one-dimensional variant of the \( F_7 \) test function is approximated using GRBF and GGDL. .......... 80

6.6 Training set error trends for the GRBF and GGDL approximations in Figure 6.5. .......................................................... 82

6.7 Test set error trends for the GRBF and GGDL approximations in Figure 6.5. .......................................................... 82

6.8 A randomly-sampled noisy dataset from the \( F_7 \) test function is approximated three times using the GGDL algorithm. .......... 85

6.9 Training set error trends for the GGDL approximations in Figure 6.8. 86

6.10 Test set error trends for the GGDL approximations in Figure 6.8. 86

6.11 A randomly-sampled noisy dataset from the \( F_1 \) test function is approximated three times using the GGDL algorithm. .......... 88

6.12 Training set error trends for the GGDL-CV approximations in Figure 6.11. 89

6.13 Test set error trends for the GGDL-CV approximations in Figure 6.11. 89

6.14 Comparison of GGDL-CV, GGDL-R, SVM, and GPM for the Abalone dataset. .......................................................... 91

6.15 Comparison of GGDL-CV, GGDL-R, SVM, and GPM for the Boston dataset. .......................................................... 92

6.16 Comparison of GGDL-CV, GGDL-R, SVM, and GPM for the CPU Small dataset. .......................................................... 93

6.17 Comparison of GGDL-CV, GGDL-R, SVM, and GPM for the Diabetes dataset. .......................................................... 94

6.18 Geometry and boundary conditions for thermal driven cavity flow problem. 98

6.19 Visualization of of PIV data for a single time step. ................. 102

6.20 Full and gappy data for the first and second velocity components over three time steps: 1, 5, and 10. ......................................... 103
6.21 Comparison of GPOD, Kriging, and GGDL-R predictions of a single time step of gappy PIV data. ........................................ 105
6.22 Probability density function (PDF) of test errors in the approximation of the second velocity component by GPOD, Kriging, and GGDL-R. .... 106

7.1 Example of gradient-enhanced dataset from the \( y = \text{sinc}(x) \) function. . . 120
7.2 The synthetic dataset from Figure 7.1 is approximated using the GRBF, GRBFH-P, and GRBFH-A algorithms. ........................................ 121
7.3 A comparison of training set error for GRBFH-P and GRBFH-A approximations of the dataset in Figure 7.1. ........................................ 122
7.4 A comparison of test set error between GRBF, GRBFH-P, and GRBFH-A approximations of the dataset in Figure 7.1. ................................. 122
List of Symbols and Acronyms

α  vector of weights
δ  error bound/tolerance
λ  regularization parameter
φ  basis function
φ  basis function vector (value of basis function for collection of input vectors)
σ  basis function shape parameter
A  dictionary matrix
c  vector corresponding to radial basis function centre
D  dictionary of basis functions
H  Hilbert space
K  Gram matrix
m  number of examples in dataset
n  number of dimensions in dataset
Q  Q matrix in QR factorization
r  residual vector
R  R matrix in QR factorization
t  greedy algorithm iteration count
T  set of training data
V(·,·) loss function
w  vector of weights
x  input vector
X  input space
y  output value
$y$ output vector (collection of all $y_i$ in a dataset)

$\mathcal{Y}$ output space

CV cross-validation

GGDL greedy algorithm for general dictionary learning

GGDL-CV greedy algorithm for general dictionary learning with cross-validation

GGDL-R randomized greedy algorithm for general dictionary learning

GPM Gaussian process model

GRBF greedy algorithm for RBF approximation

GRBFH gradient-enhanced greedy algorithm for RBF approximation

GRBFH-A “active” gradient-enhanced greedy algorithm for RBF approximation

GRBFH-P “passive” gradient-enhanced greedy algorithm for RBF approximation

LASSO least absolute shrinkage and selection operator

LOO leave-one-out error

MKL multiple kernel learning

OMP orthogonal matching pursuit

ORMP order recursive orthogonal matching pursuit

RBF radial basis function

RKHS reproducing kernel Hilbert space

RMS root mean squared

SVM support vector machine
Chapter 1

Introduction

1.1 Motivation

In the design of engineering systems, numerical simulation tools are often used to predict and improve performance prior to constructing and testing physical prototypes. Despite recent advances in numerical methods and computational hardware, the main challenge associated with simulation tools stems from their increasing fidelity and dimensionality. This means the direct application of optimization tools is often computationally expensive and impractical. A promising technique for addressing this challenge is the use of surrogate models, which are approximations of the high-fidelity computational models that can be evaluated cheaply, ideally at costs which make applications like real time visualization tractable.

In this thesis we are interested in constructing surrogate models using a set of scattered observational data. The problem of constructing an approximation model using a set of scattered observational data arises in many application areas, including machine learning (Vapnik, 1998; Bishop, 2007), forecasting and control of dynamical systems (Ahmed et al., 2010), numerical methods for partial differential equations (Golbabai & Nikpour, 2015; Kadalbajoo et al., 2015), design optimization (Keane & Nair, 2005; Forrester et al., 2008), inverse problems (Kumar et al., 2010), stochastic modelling (Brandyberry, 2008) and computer graphics (Beatson et al., 2011). Approximation models constructed using an observational or training dataset are also commonly referred to in the literature as surrogate models, emulators, metamodels, or predictors. The field of engineering design optimization in particular has benefited greatly from approximation models and motivates many of the new research directions (Viana et al., 2014).

The task of constructing a surrogate/metamodel/predictor can be viewed as approximation through learning – using an available dataset to “teach” a model how to predict
output values for various input values by tuning model-specific parameters. A wide range of numerical methods can be found in the literature for modelling input-output datasets, for example, Kriging (Gunes et al., 2006) (Sacks, 1989), radial basis functions (Wendland, 2004), Gaussian process models (Rasmussen & Williams, 2006), support vector machines (Vapnik, 1998) and neural networks (Bishop, 2007). In Chapter 2 of this thesis, some of the aforementioned numerical methods are discussed in detail.

Surrogates are typically constructed from a combination of building blocks known as basis functions – specific examples include the Gaussian radial basis function with a pre-specified width, or a first order polynomial. More generally, the problem of constructing a predictor from a collection of basis functions is the problem of learning from a dictionary. Given a scattered set of training data and a dictionary of basis functions, dictionary learning attempts to find the combination of basis functions which not only does a good job of approximating the training set, but also generalizes well to unseen data – that is, the predictor must provide accurate predictions at data points which were not used during training. The major difference between popular approximation techniques lies in the way that the “best” combination of basis functions is found. Support vector machines, for example, solve a quadratic programming problem, while Gaussian process modelling uses maximum likelihood estimation to determine the best combination. Because these problems are computationally expensive for large datasets and/or large dictionaries, research in the field is mostly concerned with two topics: (i) developing modelling techniques that can scale well to large, high-dimensional datasets and (ii) developing techniques to improve the generalization ability of the approximations for a wide range of problems (Frank et al., 2016).

In many approximation problems, it is also desirable to build accurate surrogates using only a small number of basis functions. These sparse surrogates offer a reduction in the cost of prediction, making them a much more practical candidate for the direct application of optimization or visualization tools. At the same time, sparse surrogates may also generalize better to unseen data because their lower complexity prevents them “overfitting” the training data (Rubinstein et al., 2010). Regularization, covered in Chapter 2, is the most popular technique for improving surrogate generalization while also inducing sparsity. However, greedy algorithms have also demonstrated great potential for sparse dictionary learning (Temlyakov, 2011).

In this thesis we develop a novel sequential greedy algorithm for dictionary learning in an effort to build sparse and accurate surrogate models while reducing the computational and memory costs associated with training.
1.2 Thesis Objectives

The major objectives of this thesis are:

- We wish to develop efficient algorithms for sparse dictionary learning using a parameterized dictionary – that is, a dictionary whose elements (basis functions) are dependent on individual parameters.

- We wish to develop algorithms for scattered data approximation with a heterogeneous dictionary of basis functions – that is, a dictionary which need not be constrained to basis functions of any one type.

- Most greedy dictionary learning algorithms are intrinsically sequential, making them intractable for massive dictionaries. We wish to reduce the costs associated with greedy dictionary learning by looking to parallelize or randomize the procedure.

- We wish to investigate methods for adapting our greedy approximation algorithms to gradient-enhanced datasets.

- We wish to demonstrate how our greedy algorithms may be adapted to tackle other sparse approximation problems, most notably the problem of constructing sparse polynomial chaos expansions for uncertainty quantification.

1.3 Thesis Outline

In Chapter 2 we present a collection of background material related to our research. We begin with a discussion of scattered data approximation. This includes dataset notation, an overview of machine learning, and detailed descriptions of some popular techniques. The chapter then follows with a discussion of sparse dictionary learning approximation, including the classic greedy algorithm and the orthogonal matching pursuit algorithm. This chapter lays the groundwork of concepts upon which we will develop our novel greedy algorithms.

In Chapter 3 we present our first algorithm: a greedy algorithm for radial basis function approximation (GRBF). Motivated by two significant drawbacks of classical radial basis function (RBF) approximations – high computational cost and difficulties associated with finding an ideal value of the global shape parameter – we develop a novel approach which uses a greedy algorithm to learn from a parameterized dictionary and construct sparse RBF approximations with locally-variable shape parameter values.
Numerical results for GRBF, including both illustrative examples and studies for cost and accuracy, are presented in Chapter 4.

In Chapter 5, we present our second algorithm: a greedy algorithm for learning with a general dictionary (GGDL). This algorithm is motivated primarily by a desire to use a general, or “heterogenous”, dictionary of basis functions with varying shape parameters, basis function centres, and even kernel types. We again employ a greedy procedure and outline two variations of the algorithm: a novel parallel cross-validation based approach (GGDL-CV) and a randomized approach for approximating larger datasets with larger dictionaries (GGDL-R). Numerical results for GGDL, including comparisons to other machine learning methods, are presented in Chapter 6. A pair of case studies is also presented, and demonstrate: (i) how the GGDL algorithm may be applied to construct sparse polynomial chaos expansions for uncertainty quantification problems and (ii) how the GGDL algorithm may be applied to the gappy data approximation problem.

Chapter 7 extends our greedy algorithms from Chapters 3 and 5 to gradient-enhanced datasets. We begin by discussing sensitivity analysis methods for obtaining gradient information in the training dataset. After showing that the popular Hermite RBF interpolation approach suffers from the same drawbacks as classical RBF approximation, we present both a passive and an active approach for using our greedy algorithms to approximate these datasets. Numerical results are presented in the same chapter.

Finally, we use Chapter 8 to summarize the contributions of this thesis and outline a collection of directions for future work. The future work is presented as a list of fruitful research questions.
Chapter 2

Background and Related Work

In this chapter we present background material related to our thesis research. Section 2.1 covers scattered data approximation, including dataset notation and popular supervised machine learning techniques. We cover, in detail, two techniques – radial basis function expansions and kernel methods – which are relevant to the new algorithm contributions we present in Chapters 3, 5, and 7. Section 2.2 covers sparse dictionary learning. In this section we motivate the need for sparsity and describe a number of techniques for building sparse dictionary representations, including the popular LASSO technique and the greedy orthogonal matching pursuit technique.

2.1 Approximation of Scattered Data

Scattered data approximation problems typically begin with a dataset of \( m \) examples, \( \mathcal{T} := \{(x_i, y_i)\} \) where \( i = 1, 2, \ldots, m \), (2.1)

where each training “example” is a pair composed of an input \( x_i \) and output \( y_i \),

\[(x_i, y_i) \in \mathcal{X} \times \mathcal{Y} . \] (2.2)

The input space is,

\[ \mathcal{X} \subseteq \mathbb{R}^n \text{ where } n \geq 1 , \] (2.3)

and output space is,

\[ \mathcal{Y} \subseteq \mathbb{R} . \] (2.4)

The output values \( y \) are typically the result of evaluating a computationally-expensive
function or conducting a costly experiment which maps an input $x_i \in \mathcal{X}$ to its corresponding output $y_i \in \mathcal{Y}$ as,

$$y(x) : \mathbb{R}^n \rightarrow \mathbb{R}.$$  \hspace{1cm} (2.5)

An approximation algorithm is then responsible for constructing $f(x)$ – a cheap approximation/predictor of $y(x)$ – which can be used to predict $y$ at any new $x$ as $f(x) \approx y(x)$.

For future use in linear algebra operations, it is also typically useful to express the training set in matrix-vector notation as

$$y = \{y_1, y_2, \ldots, y_m\}^T \in \mathbb{R}^m \quad \text{and} \quad X = \{x_1, x_2, \ldots, x_m\}^T \in \mathbb{R}^{m \times n}. \hspace{1cm} (2.6)$$

When developing approximation algorithms, we seek ways to construct predictors $f(x)$ which best approximate the true function $y(x)$. If we assume that the observational data $x_i, i = 1, 2, \ldots, m$ are $m$ realizations of a random process drawn from the distribution $P(x)$, then the best surrogate for representing the input-output relationship minimizes the following risk function:

$$\mathcal{R}(f) = \int Q(f(x), y(x)) \, dP(x),$$

where $Q(f(x), y(x))$ is a loss function to quantify the errors between $f$ and $y$ – the squared-loss $Q(y, y') = \frac{1}{2}(y - y')^2$ is a good example. This ability to approximate the true function $y(x)$ is typically referred to as generalization – that is, we seek a predictor $f(x)$ which “generalizes” well to unseen data. In practice, the statistical properties of $P(x)$ may be unknown and we thus typically evaluate generalization as the predictor’s approximation error for a set of test examples $T_{test}$:

$$T_{test} := \{(x_i, y_i)\} \text{ where } i = 1, 2, \ldots, m_{test}, \hspace{1cm} (2.8)$$

where typically $m_{test} > m$. In real world applications, the entire set of available data is used to train the predictor, and it is not until the predictor is incorporated into the design process that its quality can be judged. When developing algorithms, $T_{test}$ aids in assessing an algorithm’s ability to use only $T$ to build an approximation of $y(x)$.

Given a set of $m_{test}$ test examples as in (2.8), a number of error forms may be used, including the popular mean squared error (MSE),

$$E_{MSE}(f) = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} (y_i - f(x_i))^2,$$  \hspace{1cm} (2.9)
and the root mean squared error (RMSE),

\[ E_{RMSE}(f) = \sqrt{\frac{1}{m_{test}} \sum_{i=1}^{m_{test}} (y_i - f(x_i))^2}. \] (2.10)

If, as in (2.6), we collect the test set outputs into the vector \( y_{test} \) and the values predicted by \( f(x) \) into the vector \( y_{pred} \), we may also employ the vector 2-norm to assess generalization using the test set residual \( y_{test} - y_{pred} \) as,

\[ E_{norm}(f) = \frac{\|y_{test} - y_{pred}\|_2}{\|y_{test}\|_2}. \] (2.11)

Unless otherwise stated, this is the test set error definition we will employ throughout the remainder of this thesis to assess predictor accuracy.

The relationship between a predictor’s error over the training set and its error over the test set is perhaps best described by the bias-variance tradeoff [Friedman 1997]. We may define bias as error due to an inability to capture the underlying trends in the dataset, and variance as the error due to an over-sensitivity to small fluctuations in the training set. In general, models of low complexity (such as simple linear regression) have a high bias but low variance. That is, they are not capable of accurately modelling certain features of the dataset, but do not vary much between different samples of the training set. Models of high complexity, however, have low bias but suffer from high variance. That is, they have enough flexibility to accurately fit the training dataset with high precision, but different training set samples from the same true function \( y(x) \) significantly change predictor \( f(x) \). Predictors with a high bias tend to “underfit” the data, while predictors with a high variance “overfit” the data. Figure 2.1 provides an example of the bias-variance tradeoff as polynomial predictors attempt to approximate noisy samples of data from the one-dimensional function \( y = \text{sinc}(x) \). Polynomial predictors of three different orders are used to approximate five random samples, each of \( m = 50 \) points, from the function. The second order polynomial predictors exhibit underfitting but have a low variance. The twelfth order polynomial predictors exhibit overfitting but have a high variance. The sixth order polynomial predictors appear to find a good balance between bias and variance.

Equipped with an understanding of the scattered data approximation problem, as well as an understanding of how to assess the quality of predictors, we may now look at techniques for approximation. The field of machine learning, to which many of these techniques belong, is conceptually divided into two larger subfields: supervised learning,
Chapter 2. Background and Related Work

Figure 2.1: Polynomial predictors are used to approximate a random sample of the function $y = \text{sinc}(x)$ with $\mathcal{N}(0, 0.2)$ synthetic noise. One random training set sample is shown in (a) along with the true function (---). Order 2 polynomial predictors exhibit high bias but low variance (b). This is an example of underfitting. Order 6 polynomial predictors exhibit low bias and variance (c). Order 12 polynomial predictors exhibit low bias but high variance (d). This is an example of overfitting.

and unsupervised learning. Unsupervised learning deals with approximating densities and distributions of $\mathbf{x}$, without outputs $y$. For more on unsupervised learning, we refer the reader to Hastie et al. (2009). The problem of constructing surrogate models, as presented above for applications such as engineering design, falls into the realm of supervised learning. More specifically, we are interested in regression – where the outputs/target values are $y \in \mathbb{R}$ – as opposed to another part of supervised learning, classification – where the outputs/target values are categorical or binary. For more on classification, we refer the reader to Hand & Henley (1997). A number of regression techniques are covered
in comparative reviews such as those by Jin et al. (2001) for optimization and Meer et al. (1991) for computer vision. Below we present two of the more popular techniques – radial basis function (RBF) expansions, and kernel methods. In Chapter 3 we will build upon this presentation of RBF expansions to design our greedy RBF algorithm. Similarly in Chapter 5 we will build upon this presentation of kernel learning to introduce multiple kernel learning and our greedy algorithm for general dictionary learning.

2.1.1 Radial Basis Function Expansions

Many popular nonparametric approximation techniques involve the use of expansions in radial basis functions (RBFs); see, for example, Franke (1982); Rasmussen & Williams (2006); Bishop (2007); Vapnik (1998). The use of RBF expansions can be theoretically justified by the representer theorem in approximation theory (Wahba, 1990). RBF models are widely used in practical applications since they can be used in any arbitrary space dimension, work for scattered datasets, and allow the construction of interpolants with a specified degree of smoothness (Wendland, 2005). Hardy (1990) provided a detailed discussion of RBFs used for models in fields such as geophysics and signal processing, while Poggio & Girosi (1990) presented the relationship between RBF models and regularization networks in the field of statistical learning theory. Even though RBF models have a number of attractive features, this approach often results in large-scale ill-conditioned linear systems that can be difficult to solve efficiently. This has motivated the development of numerical schemes for tackling the ill-conditioning issue and reducing the computational complexity associated with large-scale datasets (Wendland, 2005; Beatson et al., 2011).

An RBF interpolation model in terms of positive definite basis functions (also known as Mercer kernels (Vapnik, 1998)) can be written in the form

$$f(x) = \sum_{i=1}^{m} w_i \phi_i(x, c_i, \sigma)$$  \hspace{1cm} (2.12)

where $\phi_i(x, c_i, \sigma) : \mathbb{R}^n \rightarrow \mathbb{R}$ is a radial basis function and $w = \{w_1, w_2, \ldots, w_m\}^T \in \mathbb{R}^m$ are the undetermined weights. Radial basis functions may take on a number of forms, including the popular definitions listed in Table 2.1 and illustrated in Figure 2.2. Here $c_i \in \mathbb{R}^n$ is the so-called RBF center and $\sigma \in \mathbb{R}$ is the shape parameter. These basis functions are referred to as “radial” because they operate on some measure of distance between their center $c$ and any other point in the design space, $x$. For example, the RBFs listed in Table 2.1 operate on the Euclidean norm, or $\ell_2$ norm, of $x - c$. It
is also important to note that classical RBF approximations use a total of \( m \) training data-centred RBFs, or \( \{ c_1, c_2, \ldots, c_m \} = \{ x_1, x_2, \ldots, x_m \} \).

Table 2.1: Common radial basis function definitions, including the Gaussian, multi-quadric (MQ), and inverse multi-quadric (IMQ).

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
<th>Domain of ( \sigma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian</td>
<td>( \phi_i(x, c_i, \sigma) = \exp\left(-|x - c_i|^2/\sigma\right) )</td>
<td>( \sigma \in \mathbb{R} )</td>
</tr>
<tr>
<td>Multiquadric (MQ)</td>
<td>( \phi_i(x, c_i, \sigma) = (1 + |x - c_i|^2/\sigma)^{1/2} )</td>
<td>( \sigma \in \mathbb{R} )</td>
</tr>
<tr>
<td>Inverse MQ (IMQ)</td>
<td>( \phi_i(x, c_i, \sigma) = (1 + |x - c_i|^2/\sigma)^{-1/2} )</td>
<td>( \sigma \in \mathbb{R} )</td>
</tr>
</tbody>
</table>

Figure 2.2: Gaussian (left), multiquadric (mid), and inverse multiquadric (right) radial basis functions are plotted for a range of shape parameter values. Arrow shows direction of increasing \( \sigma \). Taken from [Keane & Nair (2005)](#).

The weights can be calculated by solving the following linear algebraic system of equations

\[
Kw = y, \tag{2.13}
\]

where the Gram matrix \( K \in \mathbb{R}^{m \times m} \) is the result of evaluating \( m \) RBFs, each centred at a training point, at every other training point, or,

\[
K_{ij} = \phi_j(x_i, c_j, \sigma). \tag{2.14}
\]

Solution of the preceding linear system provides an \( u \)-weighted linear combination of \( m \) RBFs, each centred at a unique point from the training dataset.

In practice, Gaussian RBFs are most commonly used because they are positive definite, smooth and infinitely differentiable. In the algorithm we propose in Chapter \( \mathbf{3} \),
other RBFs as the multiquadric and inverse multiquadric can also be readily used with trivial changes. The main difference being that if the RBF is a conditionally positive definite basis function of order $q$, then to obtain a unique solution for the weight vector, a polynomial of order $q - 1$ needs to be included in (2.12) to guarantee invertibility of the Gram matrix [Michelli, 1986]. For noisy datasets, a regularization term is appended to the Gram matrix to ensure good generalization [Poggio & Girosi, 1990]. It should also be noted that most RBF models assume a single $\sigma$ for determining shape in all $n$ dimensions of the basis function. From a statistical perspective, this means they implicitly assume that the training output is the realization of a stochastic process with an isotropic, stationary covariance function.

### 2.1.2 Kernel Methods

Kernel learning is a technique for approximating scattered datasets which is employed to construct the popular support vector machine (SVM) predictor. Depending on the application, SVMs are also sometimes called support vector regressors (SVR) or support vector classifiers (SVC). To understand kernel learning, we must first introduce the reproducing kernel Hilbert space (RKHS), which is an example of a function space. A function space $\mathcal{F}$ is a space whose elements are functions, for example $f : \mathbb{R}^n \to \mathbb{R}$. To first define a Hilbert space $\mathcal{H}$, we will introduce definitions for the inner product and the norm:

**Definition 1.** An *inner product* $\langle \cdot, \cdot \rangle : \mathcal{F} \times \mathcal{F} \to \mathbb{R}$ is a function which, for every $a_1, a_2 \in \mathbb{R}$ and $f_1, f_2, g \in \mathcal{F}$ satisfies the following three properties:

1. Symmetry: $\langle f, g \rangle = \langle g, f \rangle$,
2. Linearity: $\langle a_1 f_1 + a_2 f_2, g \rangle = a_1 \langle f_1, g \rangle + a_2 \langle f_2, g \rangle$,
3. Positive-definiteness: $\langle f, f \rangle \geq 0$ for all $f \in \mathcal{F}$ and $\langle f, f \rangle = 0$ iff $f = 0$.

**Definition 2.** A *norm* $\| \cdot \| : \mathcal{F} \to \mathbb{R}$ is a non-negative function which, for every $a \in \mathbb{R}$ and $f, g \in \mathcal{F}$ satisfies the following three properties:

1. $\| f \| \geq 0$ and $\| f \| = 0$ iff $f = 0$,
2. $\| f + g \| \leq \| f \| + \| g \|$,
3. $\| a f \| = |a| \| f \|$.
Based on this general definition, many different norms are possible. We will, however, define the norm in the function space $\mathcal{H}$ using our definition of the inner product:

$$\| \cdot \|_{\mathcal{H}} = \sqrt{\langle \cdot, \cdot \rangle}$$

(2.15)

**Definition 3.** A *Hilbert space* is a Cauchy-complete linear space possessing an inner product structure.

**Definition 4.** A Hilbert space $\mathcal{H}$ is a *reproducing kernel Hilbert space (RKHS)* if for all $\mathbf{x} \in \mathcal{X}$ there exists some $M > 0$ such that

$$|E_{\mathbf{x}}[f]| = |f(\mathbf{x})| \leq M\|f\|_{\mathcal{H}} \text{ for all } f \in \mathcal{H}$$

(2.16)

where the evaluation functional $E_{\mathbf{x}} : \mathcal{H} \rightarrow \mathbb{R}$ evaluates each function in the space at $\mathbf{x}$.

This definition of the RKHS essentially bounds the evaluation functionals. This condition allows us to evaluate each function in $\mathcal{H}$ at all points in $\mathcal{X}$ while ensuring that the inner product exists. If $\mathcal{H}$ is a RKHS, then for each $\mathbf{x} \in \mathcal{X}$ there exists a “representer” function $k_{\mathbf{x}} \in \mathcal{H}$ which has the reproducing property:

$$E_{\mathbf{x}}[f] = \langle k_{\mathbf{x}}, f \rangle_{\mathcal{H}} = f(\mathbf{x}) \text{ for all } f \in \mathcal{H}.$$  

(2.17)

This means that evaluating any $f \in \mathcal{H}$ at any $\mathbf{x} \in \mathcal{X}$ is equivalent to taking the inner product of $f$ and $k_{\mathbf{x}}$ in $\mathcal{H}$. Or simply, given two elements $\mathbf{x}_1$ and $\mathbf{x}_2$ as well as their representers $k_{\mathbf{x}_1}$ and $k_{\mathbf{x}_2}$, we have:

$$k_{\mathbf{x}_1}(\mathbf{x}_2) = \langle k_{\mathbf{x}_1}, k_{\mathbf{x}_2} \rangle_{\mathcal{H}}$$

(2.18)

We can now define the reproducing kernel associated with $\mathcal{H}$ as follows,

**Definition 5.** The *reproducing kernel* of a RKHS $\mathcal{H}$ is a function $k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ defined as $k(\mathbf{x}_1, \mathbf{x}_2) := k_{\mathbf{x}_1}(\mathbf{x}_2)$ which is both symmetric and positive definite:

$$k(\mathbf{x}_1, \mathbf{x}_2) = k(\mathbf{x}_2, \mathbf{x}_1)$$

(2.19)

$$\sum_{i,j=1}^{n} a_i a_j k(\mathbf{x}_i, \mathbf{x}_j) \geq 0$$

(2.20)

for any $n \in \mathbb{N}$, $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathcal{X}$, and $a_1, \ldots, a_n \in \mathbb{R}$. 

Additionally, we may use the “feature map” function for each kernel, \( \psi : \mathcal{X} \rightarrow \mathcal{H} \) to express the reproducing kernel in terms of a mapping from any example \( \mathbf{x} \in \mathcal{X} \) to its representer \( k_x \). Following from (2.18), we have:

\[
k_{x_1}(x_2) = \langle k_{x_1}, k_{x_2} \rangle_{\mathcal{H}} = \langle \psi(x_1), \psi(x_2) \rangle = k(x_1, x_2)
\] (2.21)

Table 2.2 lists four popular reproducing kernels used in kernel learning. It should be immediately clear that there are some parallels between learning a radial basis function predictor (as in Chapter 3) and a kernel predictor. A Gaussian kernel with some shape parameter \( \sigma \), for example, describes a RKHS which is simply the function space composed of all \( \sigma \)-parameterized Gaussian RBFs centred within \( \mathcal{X} \).

### Table 2.2: Common reproducing kernels, including the Gaussian, polynomial, linear, and (conditionally positive definite) sigmoid.

<table>
<thead>
<tr>
<th>Kernel Type</th>
<th>Function ( k(x_1, x_2) )</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian kernel</td>
<td>( \exp \left(-|x_1 - x_2|_{\mathcal{H}}^2/\sigma \right) ) where ( \sigma &gt; 0 )</td>
<td></td>
</tr>
<tr>
<td>Polynomial kernel</td>
<td>( (x_1 \cdot x_2 + 1)^d ) where ( d \in \mathbb{N} )</td>
<td></td>
</tr>
<tr>
<td>Linear kernel</td>
<td>( x_1 \cdot x_2 )</td>
<td></td>
</tr>
<tr>
<td>Sigmoid kernel</td>
<td>( \tanh(x_1 \cdot x_2/a + c) ) where ( a, c \in \mathbb{R} )</td>
<td></td>
</tr>
</tbody>
</table>

Having defined the RKHS and the concept of representer functions in \( \mathcal{H} \) corresponding to examples in \( \mathcal{X} \), we now note that kernel predictors from \( \mathcal{H} \) may overfit the training dataset. To circumvent this, we introduce Tikhonov regularization. As we discussed earlier in this chapter, the bias-variance tradeoff is one of the most important concepts in machine learning. Simple models with fewer parameters often do not do as well at fitting the training data as more complex models with many parameters. However, the simpler models are less likely to “overfit” the training data.

One method through which an algorithm may find the best tradeoff between bias and variance is regularization. The classical regularized risk formulation of the regression problem seeks to find a predictor \( f \), within a hypothesis space \( \mathcal{H} \), which generalizes well to unseen data examples. The risk minimizing predictor has the form,

\[
f^* = \arg \min_f R(f) + \lambda \Omega(f),
\] (2.22)

where \( R(f) \) is an empirical risk of the predictor, \( \Omega(f) \) is a regularizer, and \( \lambda > 0 \) is a regularization strength parameter. In Tikhonov regularization (Golub et al., 1999), the empirical risk is typically a sum of a convex loss function \( V : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \) over all training
examples, while the regularizing penalty is the norm in the function space $\mathcal{H}$:

$$f^* = \arg \min_{f \in \mathcal{H}} \frac{1}{m} \sum_{i=1}^{m} V(y_i, f(x_i)) + \lambda \|f\|_{\mathcal{H}}^2. \tag{2.23}$$

Common examples include the squared-loss $V(y, y') = \frac{1}{2} (y - y')^2$ for regression, or the hinge-loss $V(y, y') = \max(1 - yy', 0)$ in support vector machine classification. Tikhonov regularization gives us a way to strike a balance between empirical error and model complexity. By noting that both the squared-loss and hinge-loss are convex, we can see that using either one of these loss functions results in the objective of (2.23) also being strictly convex. We can thus conclude that it has exactly one global minimum.

We now bring attention to the fact that $\mathcal{H}$ may be infinitely-dimensional, making it practically impossible to represent a kernel predictor from this function space. To circumvent this problem and build predictors using a finite amount of storage, we make use of the representer theorem (Schölkopf et al., 2001; Argyriou et al., 2009). The representer theorem tells us that the minimizing predictor of (2.23) can be written as a kernel expansion in $\mathcal{H}$ over the training examples $\{x_1, x_2, \ldots, x_m\}$. It therefore has the form:

$$f(x) = \sum_{i=1}^{m} w_i k(x_i, x), \quad x_i \in \mathcal{X}, \quad w_i \in \mathbb{R} \tag{2.24}$$

In this risk minimization problem, we seek $f \in \mathcal{H}$ where $\mathcal{H}$ is the RKHS associated with a single kernel function $k$. In Chapter 5 we consider an extension of this problem for the case of more than one kernel function. Additional information on learning with kernels can be found in Schölkopf & Smola (2002) and Steinwart & Christmann (2008).

### 2.2 Sparse Dictionary Learning

When building approximations for a set of scattered data, it is often desirable to construct predictors which retain the desired levels of accuracy but consist of as few basis functions as possible. One clear benefit of this “sparsity” is a decreased cost associated with prediction – evaluating the predictor $f$ at new examples $x$. In addition, inducing sparsity is a variation of regularization in the predictor. Predictors with fewer basis functions have the added potential of lower variance and are less likely to overfit the training data. This desire for sparsity is echoed in a number of fields of research, including compressive sampling (Candes & Romberg, 2007), uncertainty quantification (Doostan & Owhadi, 2011), and machine learning (Tipping, 2001).
A very powerful approach for constructing sparse approximations with large scale problems is sparse dictionary learning (SDL). SDL is a popular technique for applications such as medical imagining (Lee et al., 2011) and image de-noising (Dong et al., 2011). In the context of scattered data approximation, SDL seeks to approximate the observed vector \( y \in \mathbb{R}^m \) by attempting to find a basis function “dictionary” matrix \( A = [\phi_1, \phi_2, \ldots, \phi_d] \) where \( \phi_i \in \mathbb{R}^m \) is the vector obtained by evaluating the basis function \( \phi_i \) at all \( m \) training points, and a weight vector \( w \in \mathbb{R}^d \) by solving the following problem,

\[
\min_{A, w} \| y - Aw \|_2^2 + \lambda \| w \|_0 ,
\]

where \( \| w \|_0 \) is the \( \ell_0 \) norm – the number of non-zero elements in \( w \).

In practice, the dictionary of columns \( A \) is pre-specified such that the problem of finding a sparse approximation is equivalent to minimizing the \( \ell_0 \) norm of the representation vector, also known as the weights or coefficients, \( w \). By fixing a tolerance on the least-squares error, the problem may instead be written as

\[
\begin{align*}
\text{Minimize:} & \quad \| w \|_0 \\
\text{Subject to:} & \quad \| y - Aw \|_2 \leq \delta ,
\end{align*}
\]

This problem ensures that the sparsity of the approximation is minimized, while meeting the constraint of the least-squares system within a tolerance, \( \delta \), typically found using cross-validation (Bengio & Grandvalet, 2004).

However, identifying the global minimum solution to the \( \ell_0 \) minimization problem for an over-specified set of basis functions in the dictionary matrix is an NP-hard combinatorial problem (Natarajan, 1995). For this reason, alternative approaches are used to relax the optimization problem in order to approximate the sparsest representation for the solution. Two popular methods for determining the coefficients of a sparse approximation are an incremental orthogonal greedy algorithm or a similar optimization problem that minimizes the convex \( \ell_1 \) norm for the coefficients, \( w \). The following sections will describe two algorithms for these optimization problems.

### 2.2.1 \( \ell_1 \) Minimization

One approach for generating a sparse approximation is based on minimizing the \( \ell_1 \) norm of the coefficients. Different formulations for the \( \ell_1 \) minimization problem include the unconstrained least-squares problem, and the constrained LASSO problem that is equivalent to Basis Pursuit Denoising (BPDN) under certain conditions (Donoho et al., 2006).
While the regularized least-squares adds a weighted penalty to the minimization of the residual to account for the $\ell_1$ norm of the coefficients, the LASSO problem finds the solution to the problem

$$
\min_{w} \| y - Aw \|_2^2 \text{ subject to } \|w\|_1 \leq \tau
$$

(2.28)

where the parameter $\tau$ controls the constraint on the sparsity of the approximation. As a result, by varying $\tau$, a pareto curve of possible solutions is identified that balances the training error and the $\ell_1$ norm (sparsity) of the coefficients. This implementation is then solved with quadratic programming methods including: interior-point, active set and gradient projection algorithms.

Recently, these algorithms have also been extended to iterative $\ell_1$ minimization schemes based on the re-weighting of the magnitudes for the coefficients to enhance the sparsity of the solution (Yang & Karniadakis, 2013). In addition, basis expansion techniques have also been applied to limit the terms in the expansion to the dimensions with significant coefficients (Jakeman et al., 2015). However, both methods continue to suffer from the large memory requirement of storing the matrix $A$, particularly for high-dimensional problems.

### 2.2.2 Greedy Algorithms

The greedy algorithm is a step-by-step method of problem solving which makes the locally optimal choice at each stage. Algorithm 1 outlines a general greedy procedure for the problem of maximizing a set function. At each iteration, the element with the individual largest contribution is considered as the best available addition to the solution set. The algorithm halts when adding the best available element would fail to increase $f$.

Greedy algorithms have shown great potential in applications such as sparse approximation (Tropp, 2004), signal compression (Tropp et al., 2006), and pattern recognition (Nair et al., 2003). At the same time, it has been shown that they may be applied in a variety of ways to significantly cut computational and memory costs while developing models which are much more accurate (Temlyakov, 2011).

We know that for the dictionary learning problem at hand, performing an exhaustive search over all combinations of basis functions to approximate the model response and minimize sparsity is NP-hard (Natarajan, 1995). The greedy pursuit algorithm may be applied to design a sequential procedure of incrementally selecting terms from the dictionary, $A$ and moving them to the solution matrix $K$. Given the iterative nature of the algorithms developed in this work, we use the superscript notation to denote the
Algorithm 1 General greedy algorithm for simple set function maximization problem.

**Inputs:**
Set of integers, $A = \{1, 2, \ldots, n\}$,
Real-valued function defined on all subsets of $A$, $f(A)$.

**Outputs:**
Set of chosen integers, $B$,
Resulting value of $f(B)$.

1. $t \leftarrow 0$, $B^{(0)} \leftarrow \emptyset$

2. $t \leftarrow t + 1$

3. Choose best available element in $A$: $i^{(t)} = \arg \max_{j \in A} f(B^{(t-1)} \cup j)$

if $(f(B^{(t-1)} \cup i^{(t)}) > f(B^{(t-1)}))$ then

4. $A \leftarrow A / i^{(t)}$

5. $B^{(t)} = B^{(t-1)} \cup i^{(t)}$

Return to step 2

else

Quit

end if

$k$th iteration of a vector or matrix as $b^{(k)}$ and $A^{(k)}$, respectively. At each iteration $t$, the element that is most strongly correlated with the previous residual, $r^{(t-1)} = K^{(t)} w^{(t)} - y$, is selected to result in the greatest improvement for the approximation by solving the optimization problem

$$
\phi^{(t)} = \arg \max_{\phi_j \in A} \phi_j^T r^{(t-1)}
$$

(2.29)

where $r^{(t-1)}$ is the residual at iteration $t - 1$. Different variants of the greedy pursuit algorithm have been implemented in the signal processing community including Matching Pursuit (Friedman & Stuetzle, 1981; Mallat & Zhang, 1993), Orthogonal Matching Pursuit (OMP) (Chen et al., 1989; Doostan & Owhadi, 2011), Weak OMP (Tropp, 2004), Regularized OMP (Needell & Vershynin, 2010), Compressive Sampling Matching Pursuit (Needell & Tropp, 2009), Stagewise OMP algorithm (Donoho et al., 2012), and Subspace Pursuit (Dai & Milenkovic, 2009). For the purpose of this thesis, we will consider the OMP algorithm as a benchmark for the greedy pursuit algorithm. Recent research into OMP has found that the algorithm has a number of theoretical bounds on its generalization ability (Hussain et al., 2011; Davenport & Wakin, 2010).

In OMP, after a basis function is selected, the coefficients for the approximation are calculated by performing a least-squares regression to minimize the $\ell_2$ norm of the
residual and ensure the residual is orthogonal to the selected basis functions. The residual is then updated for the approximation and the steps are repeated until the residual error tolerance, $\delta$, is achieved. A detailed implementation for the OMP algorithm is presented in Algorithm 2.

**Algorithm 2** Orthogonal matching pursuit (OMP) for sparse dictionary learning.

**Inputs:**
- Response vector $y \in \mathbb{R}^m$.
- Dictionary matrix of candidate columns $A = [\phi_1, \phi_2, \ldots, \phi_d]$ where $\phi_i \in \mathbb{R}^m$.

**Outputs** (at iteration $t$):
- Matrix of chosen columns $K^{(t)} \in \mathbb{R}^{m \times t}$.
- Coefficient vector $w^{(t)}$.

1. $t \leftarrow 0$, $K^{(0)} \leftarrow [], w^{(0)} \leftarrow [], r^{(0)} \leftarrow y$
2. Perform cross-validation to determine $\delta$
3. While ($\|r^{(t)}\|_2 > \delta$) do
   4. Find $\phi^{(t)}$ that minimizes residual: $\phi^{(t)} = \arg \max_{\phi_j \in A} \phi_j^T r^{(t-1)}$
   5. Update $K^{(t)} \leftarrow [K^{(t-1)}, \phi^{(t)}]$
   6. Remove $\phi^{(t)}$ from $A$
   7. Solve the system $K^{(t)} w^{(t)} = y$
   8. Update residual $r^{(t)} = K^{(t)} w^{(t)} - y$
4. End while

However, in order to implement the sequential OMP algorithm, it is first necessary to define $\delta$. While a small value of $\delta$ will improve the training set accuracy, it may result in over-fitting and reduce the sparsity for the approximation. As a result, it is necessary to select a residual error that coincides with a minimal generalization error for the approximation. In practice, this is often estimated using a $\nu$-fold cross-validation (CV) technique (Bengio & Grandvalet, 2004).

During CV, the $m$ samples are partitioned $\nu$ times into a reconstruction and validation set with $m_r$ and $m_v$ samples, respectively, such that $m = m_r + m_v$ and $m_v \approx \frac{m}{\nu}$. With this method, the OMP algorithm is used to determine the approximation with the $m_r$ samples in the reconstruction set for various tolerances, $\delta_r$, and the corresponding residual error $\delta_v$ is estimated on the validation set. Repeating this technique for $\nu$ instances of the validation set and averaging the values results in an heuristic estimate for the final error tolerance, $\delta$, where the truncation error on the validation set is minimized.

This procedure requires the repeated evaluation of the OMP algorithm to perform
cross-validation. In addition, for expansions with large numbers of basis functions in a candidate dictionary, OMP requires a search through all columns of $A$ to produce the greatest reduction in the residual error. These factors lead to increased computational complexity that scales with the cardinality of the dictionary.

Under certain conditions on the matrix $A$ and the sparsity of $y$, both $\ell_1$ minimization and OMP will correctly determine a unique solution that coincides with the optimal solution to the $\ell_0$ minimization problem (Donoho et al., 2006). However, both methods offer several disadvantages including the large computational cost of evaluating all available columns within the over-specified set for OMP, and the large memory requirement of storing the dictionary matrix, $A$, containing evaluations of the basis functions for $\ell_1$ minimization. In addition, both algorithms require the specification of the residual tolerance, $\delta$, or the equivalent parameter, $\tau$, in advance, which will affect the sparsity and accuracy of the final approximation.

### 2.3 Summary and Concluding Remarks

We began this chapter by introducing the problem of approximating a scattered set of data. After equipping ourselves with notation for the dataset and an understanding of how to assess the quality of predictors, we looked at two popular techniques from the field of supervised machine learning for scattered data approximation. Radial basis function expansions and kernel methods were both covered in detail. Research in sparse dictionary learning (SDL) was then introduced by first motivating the need for sparsity in a predictor with two key benefits – reduced prediction cost and the potential for improved generalization. The SDL problem was related to our problem of constructing surrogates for a scattered set of data, before two sparsity-inducing techniques were introduced: convex relaxation of the $\ell_0$ norm via $\ell_1$ minimization and greedy algorithms, headlined by orthogonal matching pursuit.

This chapter lays the groundwork upon which our novel greedy algorithms will be built in the forthcoming chapters. In the next chapter, for example, we will develop a greedy algorithm for efficiently constructing sparse RBF approximations which address some pitfalls of classical RBF expansions. To do this, we will borrow some results and techniques from SDL, while describing in detail the individual ingredients of our algorithm. Similarly in Chapter 5, we will build upon our discussion of kernel learning from Section 2.1.2 to introduce an extremely flexible algorithm for learning from general dictionaries.
Chapter 3

Greedy Algorithm for RBF Approximation

3.1 Motivation

The work in this chapter is concerned with the development of efficient numerical schemes for constructing RBF models for moderate to large-scale datasets such as those arising from design of computer experiments. More specifically, we develop a sequential greedy algorithm for efficiently learning from parameterized dictionaries to construct RBF approximations with spatially-variable shape parameters.

The idea of using greedy algorithms to improve the efficiency of RBF approximation is not new and has been successfully applied to speedup RBF based methods for function approximation and pattern recognition; see, for example, Schaback & Wendland (2000); Nair et al. (2003); Srivastava & Meade (2010). The orthogonal matching pursuit (OMP) algorithm (Pati et al., 1993) approximates functions recursively using a dictionary of wavelets and later Natarajan (1995) improved on OMP by introducing order recursion (ORMP). Floater & Iske (1998) applied a greedy algorithm to the problem of generating evenly distributed subsets of scattered data. It should also be noted that greedy algorithms have also been applied to RBF collocation methods for partial differential equations (Hon et al., 2003; Ling & Schaback, 2009).

There are two major drawbacks to classical RBF approximations. The first of these is the difficulty associated with determining an ideal value of the shape parameter (Kansa & Hon, 2000; Fornberg et al., 2011). Since classical RBF models contain a single global shape parameter in all the basis functions, it is possible to carry out parameter variation studies to decide on an appropriate value of \( \sigma \). Large values of \( \sigma \) lead to an ill-conditioned
Gram matrix whereas small values of $\sigma$ lead to poor generalization. In practice, it is often found that the best value of $\sigma$ lies close to the verge of an ill-conditioned Gram matrix. This dilemma is particularly prominent for randomly-structured datasets and is often referred to as the \textit{tradeoff principle} of RBF approximation (Fasshauer 2007).

Although some heuristics exist for selecting a good value of $\sigma$ (Hardy 1971; Franke 1982), they ultimately fail to yield models of exceptional accuracy. One option for selecting a good value of $\sigma$ is the Contour-Padé algorithm proposed in Fornberg & Wright (2004), which allows stable computations in the limit of nearly-flat basis functions. However, this approach is limited to only rather small datasets (roughly $m = 20$ for $n = 1$ and $m = 80$ for $n = 2$) (Fasshauer 2007). More recent algorithms such as RBF-QR (Fornberg et al., 2011) and RBF-GA (Fornberg et al., 2013) are capable of overcoming the limitations on sample size and are much more efficient. A second and more widely used option to tune $\sigma$ is by minimizing a cross-validation error metric. Rippa (1999) proposed an algorithm for selecting a “good” value of $\sigma$ through the minimization of the leave-one-out error – the error obtained by removing some point $x_j$ from $\mathcal{T}$, building a predictor, and evaluating it at $x_j$. The elements of the leave-one-out error vector $e = \{e_1, e_2, \ldots, e_m\}^T \in \mathbb{R}^m$ are given by,

\begin{equation}
    e_j = \frac{w_j}{K_{j,j}^{-1}} \text{ for } j = 1, 2, \ldots, m ,
\end{equation}

where $w_j$ are the weights and $K_{j,j}^{-1}$ are diagonal elements of the inverted Gram matrix $K^{-1}$. It was shown that $|e|_2$ is able to mimic the trends in test error as a function of the shape parameter. Henceforth, when comparing greedy and classical RBF models, all models created using the classical RBF approximation will employ a near-optimal global shape parameter tuned using Rippa’s method. Another less commonly used option for tuning the shape parameter is to focus on maximizing sparsity (Carlson & Natarajan 1994).

Fornberg & Zuev (2007) studied the theoretical and practical advantages of RBF approximations with spatially-variable shape parameters – namely improvements in accuracy and conditioning. They observe the Runge phenomenon as a key error mechanism and note the added benefit of clustering RBF centres to improve accuracy. Although the advantages offered by RBF approximations with spatially variable or adaptive shape parameters have been numerically demonstrated in previous studies (Kansa & Carlson 1992; Flyer & Lehto 2010), at present there are no efficient algorithms for constructing such approximations for large-scale datasets. The algorithm presented in this chapter aims to fill this gap.
Chapter 3. Greedy Algorithm for RBF Approximation

The second major drawback of classical RBF approximations is cost. The cost of solving the linear system with dense symmetric coefficient matrix in Equation (2.13) is $O(m^3)$, while the memory requirement is $O(m^2)$. With training datasets containing thousands of points, the construction of an approximation will thus prove to be expensive, especially when multiple runs are carried out to select an appropriate value of the shape parameter $\sigma$. In addition, the cost of later evaluating the model at target sites, whether for optimization or visualization, will grow with the number of training examples since each example is the centre of an RBF. For this reason a sparse model (one with fewer than $m$ final RBFs) may be beneficial. An alternate avenue for reducing computational complexity is the use of compactly supported RBFs (Wendland, 2004) which results in a sparse Gram matrix. However, this approach does not lead to a sparse RBF model and all the basis functions centered on the training points will be present in the final model. The focus of the present work is on constructing RBF expansions that are sparse in the sense that the final expansion contains only a small number of basis functions. This will enable significant reductions in model training and prediction cost for a range of applications wherein many repeated runs of the approximation are required, e.g., uncertainty analysis, real-time visualization, and approximation model-assisted optimization algorithms.

The greedy algorithm proposed in this chapter is an incremental forward greedy strategy wherein a new basis function is appended to the RBF approximation at each iteration. A key difference between the proposed greedy algorithm and those found in the literature lies in the adaptive shape parameter and kernel tuning strategy that we employ to develop an RBF expansion with spatially variable basis functions. This provides a number of significant advantages over existing greedy algorithms for RBF approximation.

Firstly, this feature allows significant improvements in computational efficiency since $k$-fold cross-validation tests involving repeated applications of the algorithm to partitions of the training dataset are no longer needed to tune the shape parameters or to choose the kernel type. Secondly, since we use a separate shape parameter for each basis function, our approach provides exceptional modelling flexibility and is better able to approximate unstructured datasets. A similar idea was used in the greedy algorithm of Schaback and Wendland (Schaback & Wendland, 2000), wherein a sequence of RBF models are constructed via repeated residual fitting in an inner-outer loop and each model in the sequence is permitted to use a different value of the shape parameter. This approach, while exceptionally efficient, suffers from slow training error convergence.

Another distinguishing feature of the algorithm proposed in this chapter is the use of an incremental thin QR factorization to update the RBF approximation. This strategy ensures numerical stability while providing significant reductions in computational com-
plexity and memory requirements compared to greedy algorithms that require the full Gram matrix to be computed \textit{a priori} and stored \cite{Chen1991, Natarajan1995}. Additional benefits of the thin QR factorization update scheme include the ability to efficiently compute the weights of all the basis functions selected so far at each iteration and the reciprocal condition number of the coefficient matrix.

The following section introduces the proposed algorithm. Numerical studies for this algorithm are showcased in Chapter 4.

### 3.2 Greedy Algorithm for RBF Approximation

In the previous section we discussed the motivation behind building sparse models with spatially variable shape parameters. We now present the general problem statement we wish to solve using sparse RBF approximations that use a unique shape parameter value for each basis function in the expansion.

Similar to \eqref{eq:2.12}, we wish to learn a sparse predictor of the form

\begin{equation}
    f(x) = \sum_{i=1}^{m} \gamma_i w_i \phi_i(x, c_i, \sigma_i),
\end{equation}

where \( \gamma \in \{0, 1\} \) is a binary variable which gives us direct control over the number of basis functions in the predictor, defined as \( \sum_{i=1}^{m} \gamma_i \). We next define a parameterized dictionary of \( m \) basis functions centered at the \( m \) training data examples,

\begin{equation}
    D(\sigma) := \{ \phi_1(x, c_1, \sigma_1), \phi_2(x, c_2, \sigma_2), \ldots, \phi_m(x, c_m, \sigma_m) \},
\end{equation}

where \( c_i \in \mathbb{R}^n \) and \( \sigma_i \in \mathbb{R}^+ \) denote the fixed center and shape parameter of the \( i \)th basis function, respectively. Finally, we reorganize the weights as \( \alpha := \gamma w \), such that a predictor built using \( t < m \) of the basis functions in \( D \) has \( t \) non-zero values in the new vector of weights \( \alpha \in \mathbb{R}^m \) (since \( \sum_{i=1}^{m} \gamma_i = t \)) and the form,

\begin{equation}
    f(x) = \sum_{i=1}^{t} \alpha_i \phi_i(x, c_i, \sigma_i).
\end{equation}

The problem statement we consider can now be stated as follows:

\textit{Given a training set} \( \mathcal{T} \), a parametrized dictionary \( D(\sigma) \), and target training error \( \delta \), find the smallest subset of \( t \) basis functions \( \phi_i, i = 1, \ldots, t \) from \( D \), constants \( \alpha_i, i = 1, \ldots, t \),
and shape parameters $\sigma_i, i = 1, \ldots, t$, if they exist, such that

$$\sum_{i=1}^{m} \left( \sum_{j=1}^{t} \alpha_j \phi_j (x_i, c_j, \sigma_j) - y_i \right)^2 \leq \delta.$$  \hfill (3.5)

The above problem statement can be rewritten as a constrained $\ell_0$ norm minimization problem of the form:

$$\text{Minimize: } ||\alpha||_0, \quad \text{Subject to: } ||K(\sigma)\alpha - y||_2 \leq \delta,$$  \hfill (3.6)

$$\text{Minimize: } ||K(\sigma)\alpha - y||_2 + \lambda ||\alpha||_1, \quad \text{where the regularization parameter } \lambda \text{ dictates the sparsity of the optimal solution.}$$  \hfill (3.8)

There are a number of specialized algorithms in the literature for solving the preceding problem; see, for example, (Tropp 2006) and (Donoho et al. 2006). However, existing algorithms cannot be used to handle a parameterized dictionary such as $D(\sigma)$ since the optimization problem is no longer convex. We thus turn to a sequential forward greedy stepwise procedure similar to orthogonal matching pursuit (see Section 2.2.2), which is an efficient strategy to solve this class of problems (Nair et al. 2003). Algorithm 3 outlines the steps of a general sequential forward greedy RBF approximation procedure, with the addition of step 5 that allows for the use of separate shape parameter values for each basis function. In the algorithm description, we use the colon notation to represent the $j$th column or row of the matrix $A$ as $A(:,j)$ and $A(j,:)$, respectively.

It can be noted from the steps outlined in Algorithm 3 that in order to design a general
Algorithm 3 General outline of greedy RBF approximation.

**Inputs:**
- Training dataset \( T \),
- Dictionary of radial basis functions \( D \),
- Stopping criteria (\( t_{max} \), \( \delta_{min} \), etc).

**Outputs** (at iteration \( t \)):
- Indices of chosen basis functions \( \mathcal{I}^{(t)} \),
- Tuned shape parameters \( \sigma_1, \ldots, \sigma_t \)
- Vector of weights \( \alpha^{(t)} \).

\[
1. \quad t \leftarrow 0, \mathcal{I}^{(0)} \leftarrow \emptyset, \ r^{(0)} \leftarrow y.
\]

repeat
\[
2. \quad t \leftarrow t + 1.
3. \quad \text{Choose next RBF from} \ D: \ i^{(t)} = \underset{j \notin \mathcal{I}^{(t-1)}}{\arg\max} J_j.
4. \quad \text{Update set of chosen basis functions:} \ \mathcal{I}^{(t)} \leftarrow \mathcal{I}^{(t-1)} \cup i^{(t)}.
5. \quad \text{Tune shape parameter for} \ \phi_{i^{(t)}}: \ \sigma_{i^{(t)}} = \arg\min L.
6. \quad \text{Update weight vector} \ \alpha^{(t)}.
7. \quad \text{Update residual vector} \ r^{(t)}.
8. \quad \text{Calculate stopping criterion (or criteria).}
\]

until Stopping criterion or criteria are met.

adaptive sequential greedy RBF approximation technique, a number of key questions need to first be addressed: (i) what is a suitable choice for the greedy selection criterion, \( J \), that is used to select a new basis function at each iteration in step 3__, (ii) how can we efficiently update the RBF model when a new basis function is chosen while tuning its unique shape parameter__, and (iii) what should the stopping criterion/criteria be? We address these choices in the subsections that follow.

### 3.2.1 Greedy Selection Criteria

In the present work, we choose RBFs centered on the \( m \) training inputs to construct our dictionary. To select a new basis function at the \( t \)-th iteration, we first find the training point which has the current highest residual as follows:

\[
i^{(t)} = \arg\max_j \left| r_j^{(t-1)} \right|, \ j \notin \mathcal{I}^{(t-1)}. \tag{3.10}
\]

Once the training data point with the highest residual is identified, we choose an RBF centered at this point as the next basis function. In other words, each iteration of the algorithm intrinsically addresses the current worst-approximated region of the input
space. Since all RBFs in the dictionary are centered at the training points, a sort operation on the residual vector $r^{(t)}$ is the only required operation to rank all elements within the candidate set, making selection straightforward. The selection procedure used in the proposed algorithm is significantly faster than alternative approaches wherein it is sought to identify the basis function that leads to maximum decrease in the residual error norm (Natarajan, 1995). In addition to this, the proposed approach also does not require all columns of the Gram matrix to be precomputed, thereby significantly reducing the memory requirements for large-scale datasets.

3.2.2 Model Update Strategy and Shape Parameter Tuning

Consider a sparse RBF approximation at iteration $t$. Following the form in (2.12), the predictor can be written as

$$f^{(t)}(\mathbf{x}) = \sum_{i \in \mathcal{I}^{(t-1)}} \alpha_i \phi_i(\mathbf{x}, \mathbf{c}_i, \sigma_i) + \alpha_{i(t)} \phi_{i(t)}(\mathbf{x}, \mathbf{c}_{i(t)}, \sigma_{i(t)}),$$  \hspace{1cm} (3.11)

where $\mathcal{I}^{(t-1)}$ is a set containing the indices of all the RBFs chosen from the dictionary at iteration $t-1$ and $i(t)$ denotes the index of the new RBF chosen at iteration $t$. In the present approach, we do not update the shape parameters of the RBFs chosen at previous iterations and tune only the shape parameter of the new RBF $\phi_{i(t)}$. In other words, we update the weights of all the $t$ RBFs chosen so far, along with shape parameter of the new RBF chosen at iteration $t$.

From a linear algebra perspective, adding a new RBF to the approximation at iteration $t$ is equivalent to appending a column to the Gram matrix $K^{(t-1)}$. Therefore, in a greedy stepwise procedure such as the one outlined in Algorithm 3, the Gram matrix at iteration $t$ has the following form,

$$K^{(t)} = \left[ K^{(t-1)}, K(:, i(t)) \right],$$  \hspace{1cm} (3.12)

where $K^{(t-1)}$ is the Gram matrix from the previous iteration and $K(:, i(t)) \in \mathbb{R}^m$ is the column corresponding to the new RBF $\phi_{i(t)}$ selected at iteration $t$. This new column $K(:, i(t))$ corresponds to the $i(t)$-th RBF and is found by evaluating the RBF at the

\footnote{We shall revisit this approach later in Chapter 5}
training examples $X$,\

$$K(:, \iota(t)) = \begin{bmatrix}
    \phi_i(t)(x_1, c_i(t), \sigma_i(t)) \\
    \phi_i(t)(x_2, c_i(t), \sigma_i(t)) \\
    \vdots \\
    \phi_i(t)(x_m, c_i(t), \sigma_i(t))
\end{bmatrix}. \quad (3.13)$$

To solve for the weights and shape parameter at iteration $t$, we therefore need to solve the following $(t + 1)$-dimensional optimization problem:

$$\begin{equation}
(\alpha(t), \sigma_i(t)) = \arg\min_{\beta \in \mathbb{R}^t, \sigma \in \mathbb{R}} \left\| \begin{bmatrix} K^{(t-1)}(:, \iota(t)) \end{bmatrix} \beta - y \right\|_2^2 , \quad (3.14)
\end{equation}$$

where $\alpha(t) \in \mathbb{R}^t$ are the non-zero RBF weights at iteration $t$ and naturally the new column $K(:, \iota(t)) \in \mathbb{R}^m$ is a function of the shape parameter being tuned.

It can be noted from (3.11) that the RBF weights appear linearly in the model structure and the shape parameter of the new basis function appears nonlinearly. This means that the minimization problem is separable. We can therefore use ideas from the variable projection algorithm [O'Leary & Rust, 2013] and rewrite the above optimization problem as a one-dimensional minimization problem (with a nonlinear objective function) as follows

$$\sigma_i(t) = \arg\min_{\sigma \in \mathbb{R}} \left\| K^{(t)}(\sigma) \beta(\sigma) - y \right\|_2^2 , \quad (3.15)$$

where the weight vector $\beta \in \mathbb{R}^t$ appearing within the objective function can be efficiently computed for a given value of $\sigma$ by solving the following linear least squares problem

$$\alpha(t) = \arg\min_{\beta \in \mathbb{R}^t} \left\| \begin{bmatrix} K^{(t-1)}(:, \iota(t)) \end{bmatrix} \beta - y \right\|_2^2 . \quad (3.16)$$

We solve the least squares problem in (3.16) using a thin QR factorization scheme. More specifically, we employ the classical Gram-Schmidt procedure with reorthogonalization to update the QR factorization of the Gram matrix due to its improved numerical stability [Daniel et al., 1976; Reichel & Gragg, 1990]. At iteration number $t$, the new column of the Gram matrix $K(:, \iota(t))$ is used to update the factors $Q^{(t-1)}$ and $R^{(t-1)}$, thereby leading to the updated QR decomposition $K^{(t)} = Q^{(t)} R^{(t)}$, where $Q^{(t)} \in \mathbb{R}^{m \times t}$, $R^{(t)} \in \mathbb{R}^{t \times t}$. The cost of updating the QR factorization using the present approach is only $O(mt)$ at iteration $t$.

The use of a thin QR factorization of the rectangular Gram matrix within the proposed sequential greedy algorithm also allows us to efficiently update the residual vector.
at iteration $t$ using the following result.

**Lemma 1.** When appending a column $K(:,i^{(t)})$ to the end of $K^{(t-1)}$ to yield the system $K^{(t)} \alpha^{(t)} = Q^{(t)} R^{(t)} \alpha^{(t)} = y$, the updated residual can be calculated as

$$r^{(t)} = r^{(t-1)} - \left( Q^{(t)}(:,t) \right)^T y Q^{(t)}(:,t). \quad (3.17)$$

**Proof.** The residuals at iterations $t$ and $t-1$ can be written as

$$r^{(t)} = y - Q^{(t)} \left( Q^{(t)} \right)^T y,$$

$$r^{(t-1)} = y - Q^{(t-1)} \left( Q^{(t-1)} \right)^T y.$$

Splitting $Q^{(t)}$ into $Q^{(t-1)}$ and its $t$-th column, we have

$$r^{(t)} = y - \left[ Q^{(t-1)}, Q^{(t)}(:,t) \right] \left[ Q^{(t-1)}, Q^{(t)}(:,t) \right]^T y.$$

Following through the matrix multiplication and substituting in the expression for $r^{(t-1)}$ completes the proof. \qed

As a consequence of Lemma 1, the residual at each iteration can be updated without explicitly calculating the weight vector $\alpha^{(t)}$. It is worth noting here that the residual error vector provides the prediction error of the RBF model at all training points at iteration $t$ (training error). Lemma 1 offers a cheap residual update equation which we use to solve the problem in (3.15) - since the residual error vector $r^{(t)}$ can be efficiently computed at each stage of the greedy algorithm, we minimize the $\ell_2$ norm of $r^{(t)}$ to estimate the shape parameter of the new RBF chosen at iteration $t$. An added benefit of this strategy is that it will help drive the residual down quickly compared to the simpler strategy of using a fixed value for the shape parameter of each basis function.

Another useful byproduct of the incremental QR update strategy is the possibility of estimating the reciprocal condition number of the coefficient matrix at each iteration. At later stages of the iterations, it may so happen that the new column $K(:,i^{(t)})$ may numerically lie in span$\{Q^{(t-1)}\}$, where $i^{(t)}$ denotes the index of the basis function chosen from the dictionary at iteration $t$. To circumvent this problem, we monitor the reciprocal condition number of the matrix

$$\hat{Q} := \left[ Q^{(t-1)}, K(:,i^{(t)}) \right] \|K(:,i^{(t)})\|_2 \quad (3.18)$$

at iteration $t$ using the following lemma.
Lemma 2. The reciprocal condition number of $\tilde{Q}$ is given by

$$
rcond(\tilde{Q}) = \left( \frac{1 - \| (Q^{(t-1)})^T K(:,i^{(t)}) \|_2 / \| K(:,i^{(t)}) \|_2}{1 + \| (Q^{(k-1)})^T K(:,i^{(t)}) \|_2 / \| K(:,i^{(t)}) \|_2} \right)^{1/2}.
$$

(3.19)

Proof. The reciprocal condition number of $\tilde{Q}$ is given by $rcond(\tilde{Q}) = \tau_k / \tau_1$. Since the columns of $Q^{(t-1)}$ are orthonormal, i.e., $(Q^{(t-1)})^T Q^{(t-1)} = I$, it follows that the singular values $\tau_1 \geq \tau_2 \geq \ldots \geq \tau_t$ of $\tilde{Q}$ can be computed explicitly as follows (Reichel & Gragg, 1990):

$$
\tau_1 = \left( 1 + \| (Q^{(t-1)})^T K(:,i^{(t)}) \|_2 / \| K(:,i^{(t)}) \|_2 \right)^{1/2},
$$

(3.20)

$$
\tau_i = 1, \quad 2 \leq j \leq t - 1,
$$

(3.21)

$$
\tau_t = \left( 1 - \| (Q^{(t-1)})^T K(:,i^{(t)}) \|_2 / \| K(:,i^{(t)}) \|_2 \right)^{1/2}.
$$

(3.22)

Using the above expressions for $\tau_t$ and $\tau_1$ leads to (3.19). \qed

3.2.3 Choice of Stopping Criteria

Ultimately the model’s sparsity is determined by how soon a stopping criterion is met. For improved robustness of the greedy algorithm, we propose the use of multiple stopping criteria. Apart from a training error limit and a maximum number of iterations, $t_{\text{max}}$, two other stopping criteria should be mentioned: stopping prior to encountering ill-conditioning and stopping prior to increasing the generalization error. In the proposed algorithm, the condition number of the Gram matrix is monitored at each iteration using (3.19). This allows the algorithm to be halted and backtracked to avoid an ill-conditioned system. We adopt a backtracking procedure which monitors the relative change in some generalization error metric and in the present study we use the leave-one-out error (LOO) as an estimate of the generalization error. An approximation for the leave-one-out error can be efficiently computed using the following lemma.

Lemma 3. Given $Q^{(t)} r^{(t)} a^{(t)} = y$, with residual $r^{(t)}$, the leave-one-out error (LOO) can be calculated as

$$
\text{LOO} = \frac{1}{m} \sum_{i=1}^{m} \left( \frac{\tau_i^{(t)}}{1 - Q^{(t)}(i,:) Q^{(t)}(i,:)^T} \right)^2.
$$

(3.23)

Proof. The proof is straightforward and follows from the fact that, using Lemma 1, the
residual at the $t$-th iteration is:

$$r^{(t)} = y - Q^{(t)} \left( Q^{(t)} \right)^T y$$

The general expression for the leave-one error in terms of $r^{(t)}$ can then be written as

$$\frac{1}{m} \sum_{i=1}^{m} \left( \frac{r^{(t)}_i}{\partial r^{(t)}_i/\partial y_i} \right)^2$$

(Golub & Van Loan, 1996).

Similarly, it is also possible to compute the generalized cross-validation error (GCV) (Bengio & Grandvalet, 2004) at the $t$-th iteration as follows

$$\text{GCV}(r^{(t)}, t) = 1 \sum_{i=1}^{m} \left( \frac{r^{(t)}_i}{1 - \gamma} \right)^2,$$  (3.24)

where

$$\gamma = \frac{1}{m} \sum_{j=1}^{m} Q^{(t)}(j,:)Q^{(t)}(j,:)^T.$$

Given a trial value of the shape parameter $\sigma$, LOO can be calculated by updating the matrix $Q^{(t)}$ and vector $r^{(t)}$ and subsequently using Equation (3.23). It is worth noting here that the expression for the leave-one-out error in (3.23) is exact only when the centers of the RBFs in the dictionary are independent of the training points. However, it provides a cheap and reasonable quality approximation of the brute-force leave-one-out error (which would involve leaving each training point out and reconstructing the RBF model $m$ times) and is particularly accurate when $m \gg t$. To improve the accuracy of the LOO estimate, a column should also be deleted from the QR factorization if the training point being left out coincides with the set of chosen RBFs $\mathcal{I}$. However this additional step increases the computational cost of calculating the LOO error.

Figure 3.1 illustrates the trend between training time and test error as more basis functions are used in the greedy RBF approximation for the test function $F_7$ listed in Appendix A. It can be noted that, as expected, when more RBFs are added to the greedy approximation, the training time increases. After around 125 iterations, the model starts to overfit and the test error starts increasing. In this example, the approximate LOO error is quite accurate at detecting overfitting, while the brute-force LOO error mimics the trends of the test error almost exactly. We note here that the exact values of the errors are not of concern here. It is instead the iteration-to-iteration changes which are used to find the optimal predictor sparsity by detecting an increase in the error.

We recommend using the brute-force LOO error metric for small data sets or when training cost is not a concern. In all other cases, the approximate LOO error estimate
is preferable. Consider the difference in the LOO error between successive iterations, i.e., $\Delta \text{LOO} = \text{LOO}^{(k)} - \text{LOO}^{(k-1)}$. Our numerical experiments suggest that a good indication of degradation or diminishing improvements in generalization error is either five consecutive iterations of $\Delta \text{LOO} < 10^{-12}$ or an increase in the moving average of the five most recent LOO values. In practice we calculate a moving average of the termination criterion in two recent windows of about 10 iterations, i.e., $\mu_1 = \frac{1}{10} \sum_{i=t-9}^{t} \text{LOO}^{(i)}$ and $\mu_2 = \frac{1}{10} \sum_{i=t-19}^{t-10} \text{LOO}^{(i)}$. A check to see if $\mu_1 > \mu_2$ provides a reasonable indication of either degradation or diminishing returns in generalization. Once this condition is passed, we update the model by deleting basis functions until the minimum of the LOO criterion is reached.

### 3.2.4 Final Algorithm

The steps involved in the proposed sparse greedy RBF (GRBF) algorithm that brings together all the features discussed earlier are outlined in Algorithm 2.

The final model consists of $k$ RBFs and the algorithm outputs provide the necessary information for each RBF (index of RBF centre in $D$, shape parameter, weight) to
Algorithm 4 Greedy QR procedure for learning from parameterized RBF dictionaries.

**Inputs:**
- Training dataset $\mathcal{T}$
- Dictionary of radial basis functions $\mathcal{D}$
- Stopping criteria ($t_{\text{max}}$, $\delta_{\text{min}}$, etc).

**Outputs** (at iteration $t$):
- Indices of chosen basis functions $\mathcal{I}(t)$
- Tuned shape parameters $\sigma_1, \ldots, \sigma_t$
- Vector of weights $\alpha(t)$.

1. $t \leftarrow 0$, $\mathcal{I}(0) \leftarrow \emptyset$, $r(0) \leftarrow y$, $Q(0) \leftarrow [\ ]$, $r(0) \leftarrow [\ ]$.

repeat

2. $t \leftarrow t + 1$.
3. Choose RBF centre where residual is highest: $i(t) = \arg \max_{j \not\in I(t)} |r_{j-(k-1)}|$.
4. Update set of chosen basis functions: $\mathcal{I}(t) \leftarrow \mathcal{I}(t-1) \cup i(t)$.
5. Tune shape parameter for $K(\cdot, i(t))$: $\sigma_{i(t)} = \arg \min ||r||_2$.
6. Update $Q(t-1)$ and $R(t-1)$ to $Q(t)$ and $R(t)$ with $K(\cdot, i(t))$.
7. Update residual: $r(t) \leftarrow r(t-1) - (Q(t)(\cdot, t))^T y Q(t)(\cdot, t)$.
8. Calculate stopping criteria.

until Stopping criteria are met.

9. Solve for weights from the system: $R(t) \alpha(t) = (Q(t))^T y$.

construct the final approximation as follows

$$f(x) = \sum_{i \in \mathcal{I}(k)} \alpha_i \phi_i(x, c_i, \sigma_i),$$

(3.25)

where $\mathcal{I}(k)$ contains the indices of the $k$ basis functions chosen by the greedy algorithm.

### 3.3 Tuning the RBF Kernel Type

As discussed earlier, classical RBF approximations typically rely on a single global shape parameter. In addition they are also based on a single global kernel type, e.g. Gaussian, Multiquadric, etc. In the rare case that multiple kernel types are used to approximate a dataset, they are typically combined through a simple weighted addition of two separate predictors.

The GRBF algorithm however, is flexible enough to allow for on-the-fly “tuning”,

or selection, of the RBF kernel type. The algorithm has the advantage of building a predictor by adding a single radial basis function at each iteration. Just as the shape parameter $\sigma_{i(t)}$ is tuned at iteration $t$, an altogether different expression for $\phi_{i(t)}$ may be selected. Following the outline in Algorithm 3, we earlier used $L = \|r\|_2$ as the objective function for tuning the shape parameter $\sigma_{i(t)}$ at each iteration $t$. Similarly, we may again use the $\ell_2$ norm, or some other objective function, to select between a variety of RBF expressions, including the common RBFs listed in Table 2.1.

Whereas in this section we chose columns from a Gram matrix $K$ which was assumed to have the same expression (see (2.14)), it is also possible to select columns from Gram matrices with varying kernel expressions. Because the rows of $K(:, i(t))$ at each iteration are simply $\phi_{i(t)}$ applied to the training set examples $T$ (see (3.13)), a wide variety of functions may be considered and evaluated based on their ability to minimize $L$. Once any candidate $K(:, i(t))$ is calculated, the incremental QR routines are used to temporarily update $r$, allowing us to use a variety of norms to rank the candidates.

However, it is important to note that to use the same selection criterion, any candidate RBF kernel types should still be “centred” at the RBF centre chosen in the preceding step of the algorithm. Using data-centred RBFs allowed us to define a dictionary of basis functions $D$, and at each iteration use the residual $r$ to select a candidate RBF from $D$ before tuning its shape parameter. By using a dictionary of basis functions which do not coincide with the training examples $X$, we lose the ability to quickly sort through $D$.

In order to modify the procedure to allow for non-data-centred basis functions, a different selection criterion must be implemented to quantify the benefit of choosing one basis function over another. It is due to this modification that we forego implementing a “kernel tuning” procedure in GRBF and instead opt for a much more general dictionary learning algorithm in Chapter 5. Nonetheless, it should be made clear that predictors of varying kernel types may be constructed using a modified version of the GRBF algorithm and these modifications are trivial to implement.

3.4 Summary and Concluding Remarks

In this chapter, we referred to our introduction of radial basis functions from Chapter 2 to introduce our GRBF algorithm. After establishing our notation for the training dataset $T$, we discussed RBF interpolation and its two major drawbacks: difficulties associated with determining an ideal value of the shape parameter, and high cost associated with both building a predictor and later evaluating it for optimization, visualization, or similar applications.
Motivated by these drawbacks, we presented a novel approach which uses a greedy algorithm to construct sparse RBF approximations with locally-variable shape parameters. This was done by first presenting a general outline of a greedy algorithm for RBG approximation and then discussing in detail our choices of the various “ingredients”: the greedy selection criteria, the model update strategy and shape parameter tuning, and the choice of stopping criteria. The final step-wise GRBF algorithm was presented in Algorithm 4 before a brief discussion on its ability to tune not only the individual shape parameters but also the individual RBF kernel types.

We conclude by remarking that the GRBF algorithm does more than address the specific problems of classical RBF approximations. It demonstrates a way to use a greedy step-wise procedure to “build up” a surrogate model from pre-defined building blocks, which in this case were radial basis functions. It is machine learning’s staple idea that complex surrogate models may be outperformed in both accuracy and cost by simpler alternatives (Bishop, 2007). Motivated by this, we chose to apply the greedy algorithm as our method of choice for constructing these alternatives. In the next chapter, we use numerical studies to assess the quality of predictors constructed using GBRF.
In this chapter, we showcase the performance of the greedy radial basis function (GRBF) algorithm from Chapter 3. Composed of two sections, this chapter begins with demonstrative examples and later provides detailed numerical results for the GRBF algorithm on a number of test functions.

4.1 Demonstrative Examples

In this section we demonstrate the performance of the GRBF algorithm. We first provide a number of figures to illustrate the evolution of the predictor throughout the greedy procedure. We then compare the classical RBF procedure, implemented using Rippa’s method for shape parameter tuning (see Section 3.1), to our GRBF algorithm for a simple one-dimensional dataset obtained by sampling a test function. Finally we use training residual convergence trends to compare GRBF to similar sequential greedy algorithms from work by Schaback & Wendland (2000) and Natarajan (1995).

4.1.1 GRBF Predictor Evolution

We begin by sampling the one-dimensional $F_1$ test function (see Appendix A) on the interval $[0, 1]$ using a linear spacing. We sample a total of 50 points to create the training dataset and apply the GRBF algorithm. Gaussian RBFs with spatially-variable shape parameters are used and the resulting predictors after $t = 1$, $t = 3$, and $t = 16$ iterations are shown in Figure 4.1. The shape parameters are tuned using a grid search procedure with $\sigma$ varying on the interval $[0.001, 10]$ with a log-linear scale.

The GRBF algorithm halts after $t = 16$ iteration with the predictor illustrated in Figure 4.1(c) after detecting an increase in the 5-iteration average of the leave-one-out
Figure 4.1: A linearly-spaced dataset of 50 examples is sampled from the $F_1$ test function and approximated using the GRBF algorithm. Gaussian RBFs with spatially-variable shape parameters are used. The predictor is shown for $t = 1$, $t = 3$, and $t = 16$ iterations of the algorithm ((a),(b), and (c) respectively).
error discussed in Section 3.2.3.

As the GRBF algorithm iteratively approximates the training dataset, a number of metrics are stored for both post-approximation analysis and to serve as stopping criteria during approximation. Figure 4.2 shows the training set error at each iteration of the algorithm. As expected, the training set error drops continuously throughout the GRBF procedure. With added complexity, the predictor is able to better approximate the training set.

![Figure 4.2: Training set error for each iteration of the GRBF algorithm. A linearly-spaced dataset of 50 examples is sampled from the $F_1$ test function and approximated with Gaussian RBFs and spatially-variable shape parameters.](image)

At the same time, we expect the condition number of the rectangular Gram matrix $K^{(t)}$ to increase when a new column is added to the system for each new basis function and that column approximately lies in the span of the previously chosen columns. Figure 4.3 shows the reciprocal condition number at each iteration of the algorithm.

Lastly, Figure 4.4 demonstrates the leave-one-out error (LOO) discussed in Section 3.2.3 for each iteration of the algorithm. For this problem, we use LOO as a measure of generalization and check for an increase in the 5-iteration moving average of LOO to signify a level of adequate complexity. An increase in the moving average was detected at $t = 16$ iterations, the first visible “plateau” in the LOO trend, at which point the algorithm stopped. It should be noted that LOO did not increase significantly after a certain number of iterations because the dataset has no noise and the predictor approximates the true function better with each new RBF. We note that with a larger window, it is possible that the GRBF algorithm may not have halted until $t = 30$ or later. It is thus important to select a moving average window size which is representative of the problem at hand. Our experiments have found that windows between 3 and 10 iterations in size...
4.1.2 One-Dimensional Example

We next use the one-dimensional $F_2$ function (see Appendix A) to illustrate the benefits associated with using spatially-variable shape parameters. Figure 4.5 demonstrates a randomly-sampled dataset of 50 points as well as an RBF approximation constructed using Rippa’s method and a GRBF approximation.
Both approximations consider candidate shape parameters on the interval \([0.001, 10]\). The RBF method uses Rippa’s leave-one-out technique to find a single value of \(\sigma\) for all 50 RBFs and fails to construct an accurate approximation of the data. The GRBF approximation, however, constructs an approximation with shape parameters varying from 0.001 to 0.5 and is capable of accurately approximating the data with only 11 basis functions.

In this example, two of the dataset points are randomly selected to be within \(1 \times 10^{-5}\) of each other, simulating a common scenario in scattered data approximation – clustered data. This situation is also routinely encountered in iterative RBF-assisted optimization algorithms (Keane & Nair, 2005). Due to the clustering of points, the RBF technique encounters ill-conditioning for large values of \(\sigma\) and is forced to choose a small \(\sigma\). In practice, this issue is typically circumvented by performing a data filtering step prior to the approximation. However, the process of locating points which are too close is expensive, especially for large-scale datasets in high dimensions, and raises additional issues (e.g., point elimination metrics, dataset re-normalization, loss of valuable experiment information, etc). The GRBF approximation effectively handles such datasets with no user intervention.

We will again demonstrate some GRBF approximation trends, this time by intentionally running the algorithm for 24 iterations – well past the \(t = 11\) stopping point which GRBF decided on by detecting an increase in the moving average of LOO. Figure 4.6

Figure 4.5: A randomly-sampled dataset of 50 points from the one-dimensional \(F_2\) function is approximated using both Rippa’s RBF technique and GRBF. The RBF approximation uses a single shape parameter for all 50 RBFs and performs worse than the GRBF approximation, which uses spatially-variable shape parameters with only 11 basis functions.
shows the training set error for each iteration, once again decreasing with the addition of each new RBF. Figure 4.7 shows the gradual but sporadic decrease in the reciprocal condition number of the rectangular Gram matrix $K^{(t)}$, as is expected.

Figure 4.6: Training set error for each iteration of the GRBF algorithm. A randomly-spaced dataset of 50 examples is sampled from the $F_2$ test function and approximated with Gaussian RBFs and spatially-variable shape parameters.

Figure 4.7: Reciprocal condition number for each iteration of the GRBF algorithm. A randomly-spaced dataset of 50 examples is sampled from the $F_2$ test function and approximated with Gaussian RBFs and spatially-variable shape parameters.

Figures 4.8 and 4.9 are of particular interest because they allow us to compare the performance of LOO as an approximation of generalization error. We note once again that the GRBF algorithm halted after detecting an increase in the moving average of LOO after 11 iterations. A moving window average of 5 iterations was used to detect
an increase. The LOO trend appears to suggest that after this point, the generalization error gradually increased with each new addition of an RBF. The test set error trend confirms that this was a good choice, as it is the “valley” after which the next 10 RBF additions appear to increase the test set error.

Figure 4.8: LOO for each iteration of the GRBF algorithm. A randomly-spaced dataset of 50 examples is sampled from the $F_2$ test function and approximated with Gaussian RBFs and spatially-variable shape parameters.

Finally, we visualize the GRBF approximation after $t = 24$ iterations, where we believe overfitting has developed a predictor worse than the one at the stopping point ($t = 11$). Figure 4.10 plots both predictors. It is clear that, although the $t = 11$ predictor
is by no means perfect, the latter predictor shows signs of significant overfitting around $x = 1.25$.

![Graph showing dataset, true function, and GRBF approximations](image_url)

Figure 4.10: A randomly-sampled dataset of 50 points from the one-dimensional $F_2$ function is approximated using the GRBF algorithm with varying sparsities. Unlike the approximation at $t = 24$ iterations, the approximation after $t = 11$ iterations is able to stop before overfitting.

### 4.1.3 Residual Convergence Trends

We now compare the training residual convergence offered by the proposed GRBF algorithm against both the greedy algorithm of Schaback & Wendland (2000) and the algorithm of Natarajan (1995) which we refer to as orthogonal recursive matching pursuit (ORMP). Since all of these algorithms terminate at some training error threshold, the training error convergence trends are indicative of the sparsity of the final RBF expansion available through each algorithm.

Figure 4.11 demonstrates these trends for a 500-point dataset obtained from the $F_4$ test function (see Appendix A) using Sobol sampling (Sobol, 1967). Of the three algorithms considered, GRBF exhibits the best convergence rates. Our studies suggest that the greedy strategy of Schaback/Wendland converges slower than both ORMP and GRBF, while ORMP is only at times able to outperform GRBF. When local shape parameter tuning is introduced, GRBF is consistently the algorithm with fastest convergence.

From a computational viewpoint, the greedy algorithm of Schaback & Wendland (2000) is the fastest, followed by the proposed GRBF algorithm. In contrast, ORMP requires the full Gram matrix to be computed \textit{a priori} which results in significantly higher
memory requirements and computational cost for large datasets. We should note again, however, that these two greedy algorithms cannot effectively deal with parameterized dictionaries.

4.2 Test Functions

For the following numerical studies, we impose the following bound constraints for the shape parameter optimization problem in Equation (3.15): \( \sigma \in [0.01d_{\text{min}}, 5] \), where \( d_{\text{min}} \) denotes the minimum Euclidean distance between the currently chosen RBF center and the centers of the RBFs chosen so far. The bounds used here are similar in spirit to the approach proposed in Fornberg & Zuev (2007) and the heuristic procedure used in Flyer & Lehto (2010) for setting spatially variable shape parameters in RBF approximations. An alternative approach would be to use a validation error metric to tune the shape parameter of each basis function, which we did not consider due to computational cost reasons.

In addition, we include another stopping criterion which allows us to elegantly overcome ill-conditioning issues. We terminate the iterations if the reciprocal condition number, computed using Lemma 2 in Chapter 3, is lower than a specified threshold (set to \( 10^{-12} \) in our numerical studies).
We present numerical studies for a total of nine test functions that are described in Appendix A. We consider low-dimensional test functions with \( n = 1, 2 \) as well as higher-dimensional test functions with \( n = 10, 20 \). For both cases we study the performance of the greedy algorithm when the number of training points (\( m \)) lies in two intervals: \( 10n \leq m \leq 50n \) and \( m > 50n \). For the low-dimensional test functions the training points are generated using both uniform sampling and random sampling. The results are compared against classical RBF approximations with a single shape parameter tuned using Rippa’s method. For the cases when \( n = 10, 20 \) we use a minimum discrepancy sequence to generate the training points and the results obtained using the greedy algorithm are compared to those obtained using the classical RBF approach and Gaussian process models with an anisotropic Gaussian covariance function \([\text{Rasmussen & Williams} 2006]\). We use the RMS error over a test set as the error metric to compare the accuracy of the approximations.

Figure 4.12: Accuracy comparison between the GRBF algorithm and classical RBF algorithm for the \( F_5 \) test function. Box plots demonstrate RMS errors over 100 random samples. Black stars demonstrate RMS errors over uniformly-structured samples. The final GRBF had the following random and uniform sparsities: 12.3 and 1 for \( m = 25 \), 21.4 and 19 for \( m = 49 \), 35.6 and 52 for \( m = 100 \). Test RMS error was calculated using a uniform set of 4900 points.

4.2.1 Case 1: \( n = 1, 2, \ 10n \leq m \leq 50n \)

A total of six test functions (Appendix A) were used to compare the performance of the GRBF algorithm against classical RBF models. Samples of various sizes were drawn from each test function. For each combination of test function and sample size, one uniformly
Chapter 4. Numerical Results for GRBF

distributed training sample and 50 randomly distributed training samples were modelled with both techniques. This helped gain insight into the performance of the greedy and classical models for both structured and unstructured data. A large uniform test set was used to evaluate the test error of all models. Since the number of training points is fairly small, the brute-force LOO was used as the dominant stopping criterion (see Section 3.2.3).

Figure 4.12 provides a comparison between the test error of the greedy and classical models for the test function $F_5$ with $n = 2$. The test error for modelling the uniformly sampled training points is shown separate from the box plot of errors collected over 50 randomly generated training points. It was observed that for data sets of this size and dimensionality, the greedy model’s performance is comparable to the classical method, with the added bonus of a significant sparsity advantage. In particular with datasets that are generated via random sampling, the greedy approach produces models with a smaller variance of accuracy. Tables 4.1 and 4.2 document the test error and sparsity for all six test functions and these results show similar trends.

Table 4.1: Accuracy studies for the $F_1$, $F_2$, and $F_3$ test functions in $n = 1$ dimensions and both uniform and random datasets of various small sizes. Random sample results are taken as average over 50 instances. Bold results denote GRBF outperforming classical RBF model. The average number of basis functions chosen by the GRBF algorithm is shown in brackets.

<table>
<thead>
<tr>
<th>$m$</th>
<th>uniform sample error RBF</th>
<th>uniform sample error GRBF</th>
<th>random sample error RBF</th>
<th>random sample error GRBF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>8.950E-02</td>
<td>9.839E-02 (6)</td>
<td>1.482E+00</td>
<td><strong>2.487E-01</strong> (6.5)</td>
</tr>
<tr>
<td></td>
<td>$F_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>5.280E-02</td>
<td>4.717E-02 (15)</td>
<td>2.452E-01</td>
<td><strong>2.156E-01</strong> (11.4)</td>
</tr>
<tr>
<td>50</td>
<td>1.334E-03</td>
<td>4.241E-03 (39)</td>
<td>4.832E-02</td>
<td>5.905E-02 (28.2)</td>
</tr>
<tr>
<td>10</td>
<td>7.693E-02</td>
<td>8.037E-02 (9)</td>
<td>6.008E-01</td>
<td><strong>3.224E-01</strong> (6.5)</td>
</tr>
<tr>
<td></td>
<td>$F_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>5.417E-02</td>
<td>2.570E-03 (44)</td>
<td>2.426E-01</td>
<td><strong>1.167E-01</strong> (12.2)</td>
</tr>
<tr>
<td>50</td>
<td>2.859E-06</td>
<td>4.597E-03 (40)</td>
<td>8.295E-02</td>
<td>1.149E-01 (27.9)</td>
</tr>
<tr>
<td>10</td>
<td>6.227E-03</td>
<td>1.520E-02 (9)</td>
<td>1.792E-01</td>
<td>3.597E-01 (6.6)</td>
</tr>
<tr>
<td></td>
<td>$F_3$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>3.328E-07</td>
<td>2.570E-03 (19)</td>
<td>2.965E-03</td>
<td>3.993E-02 (14.8)</td>
</tr>
<tr>
<td>50</td>
<td>2.081E-07</td>
<td>1.925E-06 (40)</td>
<td>3.934E-02</td>
<td><strong>1.859E-03</strong> (30.8)</td>
</tr>
</tbody>
</table>

4.2.2 Case 2: $n = 10, 20, 10n \leq m \leq 50n$

The next set of studies included $n = 10$ and $n = 20$ dimensional functions with dataset sample sizes of $10n \leq m \leq 50n$. Training data was sampled using the Sobol low-
Table 4.2: Accuracy studies for the $F_4$, $F_5$, and $F_6$ test functions in $n = 2$ dimensions and both uniform and random datasets of various small sizes. Random sample results are taken as average over 50 instances. Bold results denote GRBF outperforming the classical RBF model. The number of basis functions chosen by the GRBF algorithm is shown in brackets.

<table>
<thead>
<tr>
<th>$m$</th>
<th>uniform sample error</th>
<th>random sample error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RBF</td>
<td>GRBF</td>
</tr>
<tr>
<td>25</td>
<td>1.544E-01</td>
<td>8.608E-02 (24)</td>
</tr>
<tr>
<td>$F_4$</td>
<td>49</td>
<td>9.721E-02</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>5.703E-02</td>
</tr>
<tr>
<td>25</td>
<td>2.936E-01</td>
<td>2.592E-01 (1)</td>
</tr>
<tr>
<td>$F_5$</td>
<td>49</td>
<td>4.638E-01</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>2.316E-01</td>
</tr>
<tr>
<td>25</td>
<td>2.079E-01</td>
<td>2.044E-01 (2)</td>
</tr>
<tr>
<td>$F_6$</td>
<td>49</td>
<td>1.548E-01</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>1.885E+00</td>
</tr>
</tbody>
</table>

The next set of studies were performed for $n = 1$ and $n = 2$ dimensional functions with dataset sample sizes of $m > 50n$. These studies are similar to those presented earlier in section 4.2.1. However, given the size of the data and relative cost associated with calculating the brute-force LOO error measure, the approximate LOO error was instead used as the primary stopping criteria. The cost of constructing approximations using the classical and greedy RBF algorithms were also monitored. Tables 4.5 and 4.6 document the accuracy and training times for all the methods considered.
Table 4.3: Accuracy studies for the $F_7$, $F_8$, and $F_9$ test functions in $n = 10$ dimensions and Sobol-sampled datasets of various small sizes. The adaptive greedy model is compared to both the classical RBF model and the Gaussian process model (GPM). Bold result denotes GRBF outperforming the classical RBF model, bold and italic denotes GRBF outperforming both GPM and classical model. Test RMS error calculated using the next 10,000 points in the Sobol sequence. The number of basis functions chosen by the GRBF algorithm is shown in brackets.

<table>
<thead>
<tr>
<th>$m$</th>
<th>$F_7$</th>
<th>$F_8$</th>
<th>$F_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>GPM</td>
<td>RBF</td>
<td>GRBF</td>
</tr>
<tr>
<td></td>
<td>2.222E-01</td>
<td>2.094E-01</td>
<td>2.037E-01 (36)</td>
</tr>
<tr>
<td>250</td>
<td>2.040E-01</td>
<td>2.024E-01</td>
<td>2.019E-01 (7)</td>
</tr>
<tr>
<td>500</td>
<td>1.892E-01</td>
<td>1.884E-01</td>
<td>1.880E-01 (40)</td>
</tr>
<tr>
<td></td>
<td>1.606E-02</td>
<td>1.721E-02</td>
<td>1.438E-02 (50)</td>
</tr>
<tr>
<td></td>
<td>1.499E-02</td>
<td>1.939E-02</td>
<td>1.233E-02 (24)</td>
</tr>
<tr>
<td></td>
<td>1.434E-02</td>
<td>1.738E-02</td>
<td>1.316E-02 (84)</td>
</tr>
<tr>
<td>100</td>
<td>6.696E-02</td>
<td>6.079E-02</td>
<td>5.385E-02 (4)</td>
</tr>
<tr>
<td>250</td>
<td>5.656E-02</td>
<td>5.956E-02</td>
<td>5.499E-02 (5)</td>
</tr>
<tr>
<td>500</td>
<td>5.274E-02</td>
<td>5.834E-02</td>
<td>5.206E-02 (12)</td>
</tr>
</tbody>
</table>

Table 4.4: Accuracy studies for the $F_7$, $F_8$, and $F_9$ test functions in $n = 20$ dimensions and Sobol-sampled datasets of various small sizes. The adaptive greedy model is compared to both the classical RBF model and the Gaussian process model (GPM). Bold result denotes GRBF outperforming the classical RBF model, bold and italic denotes GRBF outperforming both GPM and the classical RBF model. Test RMS error calculated using the next 20,000 points in the Sobol sequence. The number of basis functions chosen by the GRBF algorithm is shown in brackets.

<table>
<thead>
<tr>
<th>$m$</th>
<th>$F_7$</th>
<th>$F_8$</th>
<th>$F_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>GPM</td>
<td>RBF</td>
<td>GRBF</td>
</tr>
<tr>
<td></td>
<td>1.697E-01</td>
<td>1.564E-01</td>
<td>1.534E-01 (65)</td>
</tr>
<tr>
<td>500</td>
<td>1.441E-01</td>
<td>1.411E-01</td>
<td>1.392E-01 (30)</td>
</tr>
<tr>
<td>1000</td>
<td>1.343E-01</td>
<td>1.294E-01</td>
<td>1.277E-01 (91)</td>
</tr>
<tr>
<td></td>
<td>1.109E-02</td>
<td>1.220E-02</td>
<td>8.521E-03 (100)</td>
</tr>
<tr>
<td>500</td>
<td>8.884E-03</td>
<td>1.112E-02</td>
<td>8.142E-03 (44)</td>
</tr>
<tr>
<td>1000</td>
<td>8.324E-03</td>
<td>1.023E-02</td>
<td>8.035E-03 (118)</td>
</tr>
<tr>
<td>200</td>
<td>4.324E-02</td>
<td>4.270E-02</td>
<td>3.963E-02 (3)</td>
</tr>
<tr>
<td>500</td>
<td>3.871E-02</td>
<td>4.266E-02</td>
<td>3.844E-02 (5)</td>
</tr>
<tr>
<td>1000</td>
<td>3.604E-02</td>
<td>4.209E-02</td>
<td>3.134E-02 (90)</td>
</tr>
</tbody>
</table>
The GRBF algorithm performed exceptionally well. It was sometimes able to provide models with better accuracy, usually in comparable time. The sparsity advantage was most significant – at most 29% of the RBFs ($F_2$, uniform $m = 500$) while in some cases using as few as 1.5% ($F_3$, uniform $m = 2000$).

Table 4.5: Accuracy and timing studies for the $F_1$, $F_2$, and $F_3$ test functions in $n = 1$ dimensions and both uniform and random datasets of various large sizes. Random sample results are taken as average over 50 instances. Bold results denote GRBF outperforming the classical RBF model. The number of basis functions chosen by the GRBF algorithm is shown in brackets. The reported training time values are an average of the 50 random instances.

<table>
<thead>
<tr>
<th>$m$</th>
<th>uniform sample error</th>
<th>random sample error</th>
<th>training time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RBF</td>
<td>GRBF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RBF</td>
<td>GRBF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RBF</td>
<td>GRBF</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>1.08E-04</td>
<td>2.25E-07 (51)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.42E-01</td>
<td>3.23E-04 (52.1)</td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td>2.42E-01</td>
<td>3.23E-04 (52.1)</td>
<td>0.25</td>
</tr>
<tr>
<td>1000</td>
<td>4.29E-06</td>
<td>1.99E-07 (55)</td>
<td>1.25</td>
</tr>
<tr>
<td></td>
<td>2.41E-01</td>
<td>4.00E-06 (50.1)</td>
<td>1.25</td>
</tr>
<tr>
<td></td>
<td>2.41E-01</td>
<td>4.00E-06 (50.1)</td>
<td>1.25</td>
</tr>
<tr>
<td>2000</td>
<td>2.65E-01</td>
<td>1.99E-07 (43)</td>
<td>6.10</td>
</tr>
<tr>
<td></td>
<td>2.38E-01</td>
<td>2.38E-06 (49.4)</td>
<td>2.27</td>
</tr>
<tr>
<td></td>
<td>2.38E-01</td>
<td>2.38E-06 (49.4)</td>
<td>2.27</td>
</tr>
<tr>
<td>500</td>
<td>1.30E-04</td>
<td>6.35E-06 (143)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.03E-01</td>
<td>1.67E-04 (109.3)</td>
<td>0.24</td>
</tr>
<tr>
<td></td>
<td>2.03E-01</td>
<td>1.67E-04 (109.3)</td>
<td>0.24</td>
</tr>
<tr>
<td>1000</td>
<td>5.21E-06</td>
<td>1.83E-07 (142)</td>
<td>1.23</td>
</tr>
<tr>
<td></td>
<td>2.02E-01</td>
<td>2.27E-05 (111.6)</td>
<td>4.39</td>
</tr>
<tr>
<td></td>
<td>2.02E-01</td>
<td>2.27E-05 (111.6)</td>
<td>4.39</td>
</tr>
<tr>
<td>2000</td>
<td>2.20E-01</td>
<td>1.53E-04 (64)</td>
<td>6.10</td>
</tr>
<tr>
<td></td>
<td>2.01E-01</td>
<td>2.48E-05 (110.2)</td>
<td>8.47</td>
</tr>
<tr>
<td></td>
<td>2.01E-01</td>
<td>2.48E-05 (110.2)</td>
<td>8.47</td>
</tr>
<tr>
<td>500</td>
<td>1.26E-04</td>
<td>1.70E-07 (37)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.84E-01</td>
<td>2.62E-07 (41.8)</td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td>1.84E-01</td>
<td>2.62E-07 (41.8)</td>
<td>0.25</td>
</tr>
<tr>
<td>1000</td>
<td>5.05E-06</td>
<td>1.92E-07 (29)</td>
<td>1.23</td>
</tr>
<tr>
<td></td>
<td>1.85E-01</td>
<td>1.88E-07 (37.7)</td>
<td>0.77</td>
</tr>
<tr>
<td></td>
<td>1.85E-01</td>
<td>1.88E-07 (37.7)</td>
<td>0.77</td>
</tr>
<tr>
<td>2000</td>
<td>2.04E-01</td>
<td>1.62E-07 (30)</td>
<td>6.10</td>
</tr>
<tr>
<td></td>
<td>1.85E-01</td>
<td>1.27E-07 (35.2)</td>
<td>1.38</td>
</tr>
</tbody>
</table>

4.2.4 Case 4: $n = 10, 20, m > 50n$

The studies conducted on this type of data included $n = 10$ and $n = 20$ dimensional functions with dataset sample sizes of $m > 50n$. Due to the size of the datasets, the approximate LOO error metric was used instead of the brute-force LOO error estimation procedure. The time required to construct the approximations (training time) were recorded to analyze the costs of the three techniques. Tables 4.7 and 4.8 document both the accuracy and training times for the methods considered.

The GRBF algorithm dominates the studies, often providing more accurate models in fractions of the time. The sparsity was usually orders of magnitude better than the classical RBF model. There was only one scenario where the GRBF algorithm failed to produce a faster solution ($F_9$, $m = 1500$), as the stopping criteria were not met until 50% of the basis functions were chosen from the dictionary. But even in this case, the GRBF algorithm provided a more accurate model with significant sparsity.
Table 4.6: Accuracy and timing studies for the $F_4$, $F_5$, and $F_6$ test functions in $n = 2$ dimensions and both uniform and random datasets of various large sizes. Random sample results are taken as average over 50 instances. Bold results denote GRBF outperforming the classical RBF model. The number of basis functions chosen by the GRBF algorithm is shown in brackets. The reported training time values are an average of the 50 random instances.

<table>
<thead>
<tr>
<th>m</th>
<th>uniform sample error</th>
<th>random sample error</th>
<th>training time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RBF</td>
<td>GRBF</td>
<td>RBF</td>
</tr>
<tr>
<td>900</td>
<td>4.27E-03</td>
<td>1.32E-02 (55)</td>
<td>2.47E-02</td>
</tr>
<tr>
<td>1444</td>
<td>2.32E-03</td>
<td>5.75E-03 (131)</td>
<td>3.17E-02</td>
</tr>
<tr>
<td>2916</td>
<td>1.85E-03</td>
<td>4.64E-03 (142)</td>
<td>3.70E-01</td>
</tr>
</tbody>
</table>

Table 4.7: Accuracy and timing studies for the $F_7$, $F_8$, and $F_9$ test functions in $n = 10$ dimensions and Sobol-sampled datasets of various sizes. The adaptive greedy model is compared to both the classical RBF model and the Gaussian process model (GPM). Bold result denotes GRBF outperforming the classical RBF model, bold and italic denotes GRBF outperforming both GPM and the classical RBF model. Test RMS error calculated using the next 10,000 points in the Sobol sequence. The number of basis functions chosen by the GRBF algorithm is shown in brackets.

<table>
<thead>
<tr>
<th>m</th>
<th>test error</th>
<th>training time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPM</td>
<td>RBF</td>
</tr>
<tr>
<td>750</td>
<td>1.675E-01</td>
<td>1.649E-01</td>
</tr>
<tr>
<td>1000</td>
<td>1.668E-01</td>
<td>1.650E-01</td>
</tr>
<tr>
<td>2000</td>
<td>1.365E-01</td>
<td>1.360E-01</td>
</tr>
<tr>
<td>4000</td>
<td>1.375E-01</td>
<td>1.358E-01</td>
</tr>
<tr>
<td>750</td>
<td>1.410E-02</td>
<td>1.511E-02</td>
</tr>
<tr>
<td>1000</td>
<td>1.387E-02</td>
<td>1.492E-02</td>
</tr>
<tr>
<td>2000</td>
<td>1.366E-02</td>
<td>1.474E-02</td>
</tr>
<tr>
<td>4000</td>
<td>1.278E-02</td>
<td>1.368E-02</td>
</tr>
<tr>
<td>750</td>
<td>5.210E-02</td>
<td>5.658E-02</td>
</tr>
<tr>
<td>1000</td>
<td>5.191E-02</td>
<td>5.634E-02</td>
</tr>
<tr>
<td>2000</td>
<td>5.113E-02</td>
<td>5.609E-02</td>
</tr>
<tr>
<td>4000</td>
<td>4.920E-02</td>
<td>5.412E-02</td>
</tr>
</tbody>
</table>
Table 4.8: Accuracy and timing studies for the $F_7$, $F_8$, and $F_9$ test functions in $n = 20$ dimensions and Sobol-sampled datasets of various sizes. The adaptive greedy model is compared to both the classical RBF model and the Gaussian process model (GPM). Bold result denotes GRBF outperforming the classical RBF model, bold and italic denotes GRBF outperforming both GPM and the classical RBF model. Test RMS error calculated using the next 20,000 points in the Sobol sequence. The number of basis functions chosen by the GRBF algorithm is shown in brackets. No results are presented for GPM at $m = 10,000$ due to high training costs.

<table>
<thead>
<tr>
<th>$m$</th>
<th>GPM test error</th>
<th>RBF test error</th>
<th>GRBF test error</th>
<th>training time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPM</td>
<td>RBF</td>
<td>GRBF</td>
<td>GPM</td>
</tr>
<tr>
<td>1500</td>
<td>1.289E-01</td>
<td>1.255E-01</td>
<td><strong>1.230E-01</strong></td>
<td>191.337</td>
</tr>
<tr>
<td>$F_7$</td>
<td>2000</td>
<td>1.277E-01</td>
<td>1.256E-01</td>
<td>1.226E-01 (9)</td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>1.295E-01</td>
<td>1.256E-01</td>
<td><strong>1.221E-01</strong> (58)</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>x</td>
<td>1.170E-01</td>
<td><strong>1.144E-01</strong> (11)</td>
</tr>
<tr>
<td></td>
<td>1500</td>
<td>8.282E-03</td>
<td>1.047E-02</td>
<td><strong>7.996E-03</strong> (27)</td>
</tr>
<tr>
<td>$F_8$</td>
<td>2000</td>
<td>8.244E-03</td>
<td>1.044E-02</td>
<td><strong>8.119E-03</strong> (17)</td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>8.156E-03</td>
<td>9.463E-03</td>
<td><strong>7.954E-03</strong> (19)</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>x</td>
<td>9.156E-03</td>
<td><strong>7.819E-03</strong> (18)</td>
</tr>
<tr>
<td></td>
<td>1500</td>
<td>3.518E-02</td>
<td>4.176E-02</td>
<td><strong>3.445E-02</strong> (750)</td>
</tr>
<tr>
<td>$F_9$</td>
<td>2000</td>
<td>3.448E-02</td>
<td>4.053E-02</td>
<td><strong>3.625E-02</strong> (12)</td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>3.352E-02</td>
<td>3.844E-02</td>
<td><strong>3.541E-02</strong> (13)</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>x</td>
<td>3.661E-02</td>
<td><strong>2.988E-02</strong> (73)</td>
</tr>
</tbody>
</table>

### 4.3 Summary and Concluding Remarks

In this chapter, we carried out a detailed set of numerical experiments to demonstrate the performance of our GRBF algorithm. We began with two one-dimensional examples to show (i) the evolution of a GRBF predictor from the first iteration through to a final solution, using an approximation for the leave-one-out error (LOO) as the dominant stopping criterion, and (ii) the benefits GRBF’s spatially-variable shape parameters offer over the single global shape parameter of classical RBF approximations. Finally, we used a collection of test functions to compare our GRBF algorithm to both classical RBF approximation and Gaussian process modelling (GPM).

Throughout our studies, classical RBF approximations were constructed with a Fortran implementation of Rippa’s approach for tuning the global shape parameter. As outlined in Section 3.1, this approach selects the shape parameter which minimizes the leave-one-out error. GPM approximations were constructed using our own efficient in-house Fortran code. The GRBF algorithm was also implemented in Fortran.
tical studies showed that for low-dimensional datasets of small size, GRBF is sometimes able to build approximations which are more accurate than the classical RBF approach, especially for randomly-sampled datasets. For high-dimensional datasets of small size, GRBF performs exceptionally well for uniformly-sampled datasets and is comparable to classical RBF approximation for randomly-sampled data. For large datasets of both low and high dimensionality, GRBF is often able to outperform the classical RBF and GPM approaches.

Our studies show that the GRBF algorithm is able to construct approximations with accuracy comparable to, or better than, the two popular approaches. Perhaps most importantly however, the GRBF predictors are typically orders of magnitude more sparse than the competing models and are often built in less time. For datasets with over \( m = 1000 \) points, GRBF typically outperformed the competition in accuracy, time, and sparsity. It should be noted that although the test functions used in this chapter are quite difficult to approximate, the performance of each algorithm is ultimately very dependent on the approximation problem at hand. The purpose of this chapter was to demonstrate that GRBF competes admirably with two popular machine learning techniques and presents an excellent choice when predictor sparsity is important.
Chapter 5

Greedy Algorithms for General Dictionary Learning

5.1 Motivation

In Chapter 3 we outlined an efficient algorithm for building RBF approximations. We now wish to generalize the algorithm by addressing a number of key topics. In this section we discuss these topics and outline the organization of this chapter.

Choosing the kernel type: The task of selecting a problem-appropriate kernel type is often not trivial. In practice, a portion of the data, called a validation set, is typically held out of training and used to evaluate candidate models. For example, when deciding between the use of Gaussian or inverse multiquadric RBFs, a number of the training examples (typically 25-50%) may be held out and used to evaluate a validation error based on which the decision is made. To avoid this expensive procedure, it is desirable for an algorithm to be able to determine the most appropriate kernel at training time without prior knowledge of the task or additional user influence.

Non-data-centred RBFs: In the RBF algorithm of the previous chapter, the basis functions were chosen to be centred at the training data examples. This offered the convenience of trivial decision making at the candidate selection step of the greedy algorithm – basis functions were centred at training sites where the residual was highest. However, it may be desirable to centre basis functions at a variety of locations within the problem domain or even outside of the problem domain. One incentive would be to minimize the effect of the Runge phenomenon, which manifests itself as high frequency effects outside of the problem domain and can lead to poor model generalization [Fornberg & Zuev].
A more common scenario which would benefit from the custom selection of basis function centres is when the dataset has varying degrees of density in the domain or simply does not contain much data.

**Mixing kernel types:** There are inherent limitations associated with using a single kernel type for regression. For example, a dataset may be best-approximated by using a combination of Gaussian and polynomial kernels due to varying oscillation and/or noise levels in certain domain regions. It is desirable for an algorithm to be able to mix kernel types to capture problem-specific trends with greater flexibility. In our problem formulation of dictionary learning, the use of mixed kernels manifests itself in learning from diverse, or “heterogenous”, dictionaries. Our discussion touched on this topic in Section 3.3, referring to it as “tuning the kernel type” of individual RBFs. Here, we would like to lift the restriction of using RBFs and open the algorithm up to basis functions of any type.

It should now be noted that to address the above limitations, the greedy RBF algorithm from the previous chapter must be modified in a number of ways. This chapter covers the development of a new greedy algorithm capable of learning from general dictionaries. The next section builds upon our description of kernel learning in Section 2.1.2 to outline multiple kernel learning, a field of study closely related to the problem statement under consideration. Section 5.3 details the ingredients of a new parallel greedy algorithm for smaller data sets, while Section 5.4 outlines a randomized algorithm specifically designed for large-scale problems.

### 5.2 Multiple Kernel Learning

In machine learning, the field of work most closely related to the idea of constructing approximations from general or “heterogenous” dictionaries is multiple kernel learning (MKL) (Lanckriet et al., 2004). Whereas the motivation outlined above is presented as an extension of our work on radial basis function approximations, research in MKL was motivated by similar issues encountered while using unique kernels in support vector machines (SVMs), notably: difficulties associated with selecting the best kernel and difficulties in practical problems which often include data from multiple heterogenous sources (Ben-Hur & Noble, 2005; Pavlidis et al., 2002).

In applying semidefinite programming to kernel matrices, Lanckriet et al. (2004) showed that combining multiple kernels can yield an approximation that is comparable
with the best individual approximation. In fact, by using multiple kernels without hyperparameter tuning, they were able to show comparable performance to single kernel models with tuned hyperparameters. Bach et al. (2004) remarked that using multiple kernels provides added flexibility for practical problems that often involve data from a variety of heterogenous sources.

In this section, we introduce multiple kernel learning and then cover modern techniques for learning with multiple kernels, including a technique from Sindhwani & Lozano (2011) which uses a greedy algorithm.

5.2.1 Regularized Risk Minimization with Multiple Kernels

When we attempt to learn with more than one kernel, we seek to learn within a sums space of reproducing kernel Hilbert spaces (RKHSs) while using a combination of kernels. We may for example have three kernels:

- a Gaussian kernel with $\sigma = 1$,
- a Gaussian kernel with $\sigma = 2$,
- and a polynomial kernel with $d = 2$.

Our goal would now be to learn the best possible predictor while using all three of these kernels. More generally, for $M > 1$ kernels, we wish to find

$$k_\theta = \sum_{i=1}^{M} \theta_i k_i,$$

while regularizing both the kernel weights $\theta = (\theta_1, \theta_2, \ldots, \theta_M)^T$ and the new block-structured map weights, $\alpha = (\alpha_1^T, \alpha_2^T, \ldots, \alpha_M^T)^T$. Naturally, an added layer of complexity is introduced as it must now be determined which kernels are relevant to the problem and how they should be combined.

Trivially, learning with multiple kernel types may be accomplished via a variety of ensemble methods. For example, the SVM algorithm may be applied to a training set for a variety of kernels to learn the individual kernel predictors (finding $\alpha_1, \alpha_2, \ldots, \alpha_M$). Later, a validation set may be used to determine the best mixing proportions of the individual single-kernel models (to solve for the kernel weights $\theta$). However, this inevitably leads to increased training costs and prevents true kernel mixing as each kernel is used to learn in isolation before they are all eventually combined in the ensemble. It is hence
desirable to develop an algorithm which is itself able to accurately choose the kernels and weights best suited to tackle specific learning tasks.

We can define $H$, the RKHS for the multiple kernel regularized risk minimization problem, as the sum space,

$$H = H_1 \oplus \cdots \oplus H_M.$$  \hfill (5.2)

Shawe-Taylor & Cristianini (2004) showed that when the kernels are combined linearly, the feature map functions (see Chapter 2) are combined as $\psi_{\theta} = \sqrt{\theta_1} \psi_1 \times \sqrt{\theta_2} \psi_2 \times \cdots \times \sqrt{\theta_M} \psi_M$ and thus the optimization problem resembles the dictionary learning problem outlined in Chapter 2 and seeks to find $f$ in $H$, by minimizing the following:

$$\inf_{\alpha, \theta} \frac{1}{m} \sum_{i=1}^{m} V\left(y_i, \sum_{j=1}^{M} \sqrt{\theta_j} \langle \alpha_j, \psi_j(x) \rangle_{H_j}\right) + \lambda \sum_{j=1}^{M} \|\alpha_j\|^2_{H_j} + \mu \omega(\theta),$$ \hfill (5.3)

where $\mu$ is a regularization coefficient and $\omega(\theta)$ is a regularizer added to penalize complex kernel mixtures and stimulate sparsity in the solution. Kloft et al. (2011) showed that non-convexity associated with the square root in the risk function can be resolved and that the training set size and regularization strength parameters can be decoupled into a single constant $C$. The same convex optimization problem is obtained by Varma & Ray (2007):

$$\inf_{w, \theta} C \sum_{i=1}^{m} V\left(y_i, \sum_{j=1}^{M} \langle \alpha_j, \psi_j(x) \rangle_{H_j}\right) + \sum_{j=1}^{M} \|\alpha_j\|^2_{H_j} \theta_j.$$ \hfill (5.4)

Problem (5.4) is commonly referred to as one-stage multiple kernel learning because the kernel function and the predictor are learned at the same time in one optimization problem. A number of techniques have been proposed to solve variants of the one-stage MKL problem. The large majority of these techniques attempt to learn sparse kernel combinations through regularization via the $l_1$ norm (as is typical with LASSO – see Section 2.2.1). Lanckriet et al. (2004) were first to outline a semidefinite programming formulation for this problem and demonstrated a quadratically constrained quadratic programming (QCQP) optimization problem for learning a linear combination of kernels. Because their solutions to the QCQP problem were exceptionally expensive for large $m$ or $M$, MKL literature has since aimed to produce more tractable algorithms. Sonnenburg et al. (2006) reformulated the QCQP problem as a semi-infinite linear program and solved it using typical SVM techniques. Rakotomamonjy et al. (2008) introduced SimpleMKL, which updates the kernel weights using gradient descent. HessianMKL was the next, more efficient, iteration of this algorithm (Chapelle & Rakotomamonjy, 2008). Finally,
Xu et al. (2010) used the group lasso explicitly to learn with multiple kernels.

It should also be mentioned that there exists a two-stage multiple kernel learning formulation, based on the theory of kernel alignment. We will forego covering the details, and briefly list the steps of two-stage MKL as follows:

1. Learn the kernel function $k_\theta$ in (5.1), and thus $\mathcal{H}$, typically through a surrogate objective function.

2. Apply kernel methods to learn the predictor $f \in \mathcal{H}$, typically using hinge loss for support vector classifiers or squared loss for ridge regression.

Similar to the greedy approach outlined in the next subsection, the MKL algorithm we present in this chapter is based on the regularized risk minimization formulation outlined here, and not on the two-stage MKL formulation. We therefore leave it to the reader to see Cristianini et al. (2002) for more on kernel alignment and Cortes et al. (2010) for more on two-stage MKL.

### 5.2.2 A Greedy Approach to Multiple Kernel Learning

When learning sparse models, there are typically two algorithmic paths: the use of regularization techniques, such as the LASSO, and the use of greedy algorithms, such as orthogonal matching pursuit (OMP). The greedy algorithm provides an alternative to regularization by controlling model complexity through a step-wise procedure which typically increases predictor complexity with each iteration. As outlined in Section 5.2.1 MKL literature has thus far focused predominantly on inducing sparsity in the risk minimization problem through regularization techniques.

Sindhwani & Lozano (2011) recently proposed a framework for sparse learning of multiple kernel models based on Group-OMP (Swirszcz et al. 2009). In order to decouple sparsity (the number of kernels used) from “smoothness” (regularization penalty on $f$), they began by noting that we can make use of $l_p$ norms of $f \in \mathcal{H}$ as:

$$
\|f\|_{l_p(\mathcal{H})} = \inf \left\{ \left( \sum_{i=1}^{M} \|f_i\|_{\mathcal{H}_i}^p \right)^{1/p} : f(x) = \sum_{i=1}^{M} f_i(x), \ x \in \mathcal{X}, \ f_i \in \mathcal{H}_i, \ i = 1, \ldots, M \right\},
$$

and also defining the $l_0$ pseudo-norm as:

$$
\|f\|_{l_0(\mathcal{H})} = \min\{\|J\| : f = \sum_{j \in J} f_j\},
$$

(5.5)
where \( \mathcal{H} \) is the RKHS of \( M \) kernels from (5.2). The regularized risk minimization problem from (2.22) can now be reformulated as the sparse kernel selection problem of the following form:

\[
\arg\min_{f \in \mathcal{H}} \frac{1}{m} \sum_{i=1}^{m} V(y_i, f(x_i)) + \lambda \|f\|_{l_0(\mathcal{H})}^2 \quad \text{subject to} \quad \|f\|_{l_0(\mathcal{H})} \leq s.
\] (5.7)

It is clear that problem (5.7) decouples the predictor sparsity (via \( \|f\|_{l_0(\mathcal{H})} \leq s \)) from any individual kernel smoothness (via \( \lambda \|f\|_{l_2(\mathcal{H})} \)). Sindhwani & Lozano (2011) note that this decoupling gives their algorithm knobs to control sparsity and smoothness, akin to the elastic net technique (Zhou & Hastie, 2005).

The general idea behind this algorithm is to iteratively select kernels from a dictionary based on their ability to decrease the training set residual with regularization. The predictor is refit at each step by combining the kernel Gram matrices and solving for the weights, and it is then used to evaluate an updated residual. For a total of \( t \) iterations, the resulting predictor is \( f \in \mathcal{H}_G \), where \( G \) is the set of indices for the chosen \( t \) kernels and thus \( \mathcal{H}_G \) is the sum space of the RKH spaces of those kernels – as in (5.2). Sindhwani & Lozano (2011) note that an error over a validation set may be monitored and used to halt. In addition, they suggest reducing computational cost by creating a random subsample dictionary of kernels at each iteration.

Thus far, we have looked into the field of multiple kernel learning and found a number of ways through which more than one kernel may be used to construct a predictor. Based on the regularized risk minimization formulation, MKL algorithms typically aim to solve quadratically constrained quadratic programming problems or semi-infinite linear programming problems.

An alternate and less-explored approach is to use greedy algorithms to solve these minimization problems. Algorithms such as that of Sindhwani & Lozano (2011) are less common in MKL but share more similarities with our previous greedy algorithm work on RBFs. By using a greedy algorithm to iteratively select kernels and expand the RKHS for the predictor, we are able to address two of the three topics with which we motivated this chapter. Unfortunately, these approaches are exceptionally expensive to implement in practice for large \( m \) or \( M \). In addition, the desire to place basis functions at structurally-sampled locations within the problem domain, or even outside it, is not satisfied by the RKHS-based methods covered thus far. To address these issues, in the following section we present our more flexible algorithm for learning from general dictionaries.
5.3 Parallel Algorithm for General Dictionaries

Building upon both our previous RBF-specific algorithm and the literature on greedy algorithms for regularized risk minimization in MKL, we propose an algorithm for building multiple kernel predictors by learning from heterogenous dictionaries of basis functions. The following subsections outline the details of this algorithm. In addition to discussing key algorithm ingredients such as the dictionary and candidate selection procedure, we also discuss using cross validation to obtain an accurate stopping criterion and leveraging the efficiency of parallel computing to make our algorithm even more tractable in practical applications.

5.3.1 General Dictionaries and Problem Statement

We begin by bridging the gap between our greedy RBF algorithm from Chapter 3 and greedy algorithms for solving MKL problems. Whereas our algorithm added a single basis function at each iteration, greedy MKL techniques such as that of Sindhwani & Lozano (2011) add an entire kernel at each iteration. The addition of a kernel is analogous to adding a set of basis functions, in this case one centred at each training example – via the representer theorem. Algebraically, this is represented by the combination of multiple square Gram matrices. We are reminded that learning with a full set of basis functions carries a cost of $O(m^3)$, as it involves computing the Gram matrix $K$ and solving the linear system $K\alpha = y$. In addition to the high cost, the restriction of using training data sites as the basis function centres does not address the second motivating point of this chapter, outlined in Section 5.1.

We previously showed that, in general, MKL techniques typically look to combine $M$ kernels as

$$k_\theta = \sum_{j=1}^{M} \theta_j k_j ,$$

(5.8)

and construct a predictor of the form:

$$f(x) = \sum_{j=1}^{M} \theta_j \sum_{i=1}^{m} w_{j,i} \phi_{j,i}(x) ,$$

(5.9)

where $\phi_{j,i}(x) = k_j(x_i, x)$ is the $i$th of $m$ basis functions from the $j$th of $M$ kernels, $\theta_j$ are the kernel weights, and $w_{j,i}$ are the weights of the $i$th basis function from the $j$th kernel. Like Kloft et al. (2011), we note that the two levels of weights can be combined as $\bar{w}_{j,i} = \theta_j w_{j,i}$. 

In contrast, we wish to develop a greedy algorithm which, instead of building a predictor by selecting candidates from a dictionary of kernels, selects individual basis functions. Unlike existing MKL techniques, we wish to learn a much more flexible sparse multiple kernel predictor of the form:

\[ f(x) = \sum_{j=1}^{M} \sum_{i=1}^{m} \gamma_{j,i} \tilde{w}_{j,i} \phi_{j,i}(x), \]  

(5.10)

where \( \gamma \in \{0, 1\} \) is a binary variable that allows us to directly control the \( \ell_0 \) norm of \( \tilde{w} \). We re-posit the MKL regularized risk minimization problem as a problem of learning from a heterogenous \( \mathcal{D} \)-indexed dictionary \(^{1}\) of \( d = Mm \) basis functions, \( \{\phi_j\}_{j \in \mathcal{D}} \) and reorganize the weights as \( \alpha = \gamma \tilde{w} \) such that the predictor now has the form:

\[ f(x) = \sum_{i=1}^{d} \alpha_i \phi_i(x). \]  

(5.11)

Similar to (3.5), we may now state our problem statement as the following:

**Given a training set** \( \mathcal{T} \), a \( \mathcal{D} \)-indexed dictionary of basis functions \( \{\phi_j\}_{j \in \mathcal{D}} \) and target training error \( \delta \), find the smallest subset of \( t \) basis functions \( \phi_i, i = 1, \ldots, t \) from \( \mathcal{D} \) and constants \( \alpha_i, i = 1, \ldots, t \), if they exist, such that

\[ \sum_{i=1}^{m} \left( \sum_{j=1}^{t} \alpha_j \phi_j(x_i) - y_i \right)^2 \leq \delta . \]  

(5.12)

The basis functions in the dictionary need not be in Hilbert spaces spanned by various kernels. Examples include:

- The representer of any \( c \in \mathcal{X} \) from a reproducing kernel (see (2.17) and (2.18)) in Table 2.2, such as the Gaussian,

\[ \phi(x) = k_c(x) = k(x, c) = \exp(-\|x - c\|^2/\sigma) \]  

(5.13)

where \( \sigma \in \mathbb{R} \) is a pre-specified shape parameter

- Any type of global polynomial function, such as,

\[ \phi(x) = x_1 + x_2^2 + \ldots \]  

(5.14)

\(^{1}\)Whereas in Chapter 3 we used \( \mathcal{D} \) as the set of candidate basis functions, here it is the set of indices corresponding to the set of basis functions. This is done for notational clarity in the algorithms to follow.
As with the greedy selection of RBFs, the addition of an arbitrary basis function $\phi_j$ is algebraically equivalent to expanding the Gram matrix $K$ with the column,

$$
\phi_j = \begin{bmatrix}
\phi_j(x_1) \\
\phi_j(x_2) \\
\vdots \\
\phi_j(x_m)
\end{bmatrix}.
$$ (5.15)

Therefore, the general dictionary we use in this algorithm is represented by a collection of columns, each computed by evaluating the associated basis function at all training points $\{x_1, x_2, \ldots, x_m\}$. Or simply, $\{\phi_j\}_{j \in \mathcal{D}}$. We can compile the general set of $d$ basis function candidates into a dictionary, or regressor, matrix $A \in \mathbb{R}^{m \times d}$ as follows,

$$
A = [\phi_1 \ \phi_2 \ \ldots \ \phi_d],
$$ (5.16)

and define a set of chosen column indices which starts empty, $\mathcal{I}^{(0)} = \emptyset$. The algorithm iteratively adds indices to $\mathcal{I}$ such that at iteration $t$, the Gram matrix is $K^{(t)} = A(:, \mathcal{I}^{(t)})$.

Our problem statement (5.12) may now be written as:

$$
\text{Minimize:} \quad ||\alpha||_0
$$

$$
\text{Subject to:} \quad ||A\alpha - y||_2 \leq \delta
$$ (5.17) (5.18)

where $\delta$ is typically selected using cross-validation. To circumvent the fact that this problem is NP-hard, we can use the idea of convex relaxation and solve instead the following unconstrained optimization problem (Zhang, 2010):

$$
\text{Minimize:} ||A\alpha - y||_2 + \lambda ||\alpha||_1,
$$ (5.19)

where the parameter $\lambda$ dictates the sparsity of the optimal solution. There are a number of specialized solvers including fast coordinate descent search algorithms for solving the preceding optimization problem (Friedman et al., 2010). However, they are not straightforward to parallelize, except possible cross-validation wherein the parameter $\lambda$ is estimated by training the model on multiple partitions of the training data. We instead wish to implement a greedy algorithm to develop an alternative approach that enables a highly efficient parallel implementation, while providing better control over the model sparsity.

The greedy selection of basis functions is represented by the column-wise expansion of a single Gram matrix — a procedure with strong connections to orthogonal matching
pursuit (OMP), as discussed in Chapter 2. To improve efficiency over the baseline OMP technique, we can once again employ the incremental thin QR factorization of the Gram matrix $K$, as in our previous algorithm for greedy RBF approximation. To develop a truly robust and accurate algorithm, we will also need to adapt the candidate selection technique and stopping criteria to the task at hand. The following subsections detail these ingredients.

5.3.2 Candidate Selection

To rank the candidate basis functions at each iteration, we consider using an expression for the future training residual. This idea is motivated by the work of Grote & Huckle (1997) and Gould & Scott (1998), who proposed a similar approach for constructing sparse approximate preconditioners of large-scale sparse coefficient matrices. Because we plan to use sparsity in the number of basis functions to control overfitting, there is no explicit need for regularization in the risk minimization problem. Given the residual at the end of the previous iteration, $r_{t-1}$, and all possible candidates $j \in D_{t-1}$, Grote & Huckle (1997) consider the following one-dimensional minimization problem:

$$
\min_{\mu_j} \|r_{t-1} + \mu_j \phi_j\|_2,
$$

which has the approximate solution:

$$
\mu_j = -\frac{\phi_j^T r_{t-1}}{\|\phi_j\|_2^2}.
$$

Therefore to quantify the benefit of adding basis function $j$, we can compute an approximation to the norm of the future residual as:

$$
\delta_{j,\text{approx}}^{(t)} = \|r_{t-1}\|_2^2 - \frac{(\phi_j^T r_{t-1})^2}{\|\phi_j\|_2^2}.
$$

However, we can make use of a projection matrix for an efficient way of calculating the exact residual decrease. Gould & Scott (1998) show that the expression for the exact residual is,

$$
\delta_j^{(t)} = \|r_{t-1}\|_2^2 - \frac{(\phi_j^T r_{t-1})^2}{\|P_{t-1} \phi_j\|_2^2},
$$
where $P$ is the projection matrix, which at iteration $t$ is defined as:

$$P^{(t)} = I - K^{(t)}((K^{(t)})^T K^{(t)})^{-1}(K^{(t)})^T.$$  \hfill (5.24)

We note that the dominant cost is calculating $P^{(t-1)} \phi_j$ using the Gram matrix $K^{(t-1)}$. Since $P^{(t)} = I - Q^{(t)}(Q^{(t)})^T$, we may make use of the QR factorization as:

$$\delta_j^{(t)} = \|r^{(t-1)}\|_2^2 - \frac{(\phi_j^T r^{(t-1)})^2}{\|\phi_j\|_2^2 - \|(Q^{(t-1)})^T \phi_j\|_2^2}.$$  \hfill (5.25)

However the additional calculation of $(Q^{(t-1)})^T \phi_j$ may still present considerable cost for even moderately sized dictionaries. Instead, Gould & Scott (1998) show that for each candidate $j$, the denominator of (5.23) can be updated iteratively as

$$\|P^{(t)} \phi_j\|_2^2 = \|P^{(t-1)} \phi_j\|_2^2 - \left[(Q^{(t-1)})^T \phi_j\right]^2,$$  \hfill (5.26)

where $Q$ is the most recent column of $Q$, or $Q^{(t)} = Q^{(t)}(:, t)$. Defining the denominator for candidate $j$ as $\varphi_j$, we can update its value at each iteration through,

$$\varphi_j^{(t)} = \varphi_j^{(t-1)} - \left[(Q^{(t-1)})^T \phi_j\right]^2,$$  \hfill (5.27)

and finally at iteration $t$ choose the index of the candidate basis function with the minimum future residual by solving,

$$i^{(t)} = \arg \max_j (\phi_j^T r^{(t-1)})^2 / \varphi_j.$$  \hfill (5.28)

This formulation will be the method through which we will rank basis function candidates at each iteration of our algorithm. It ensures that, when evaluating the potential of a candidate, only two vector-vector products are required in order to calculate the future training residual, as long as we keep track of the denominator of (5.23) for each candidate and use (5.27) to continuously update its value.

**5.3.3 Stopping Criterion: $\nu$-fold Cross Validation**

Similar to our greedy RBF approximation algorithm from Chapter 3, a number of stopping criteria can be used to terminate the algorithm while learning from a general dictionary of basis functions. The algorithm may halt if at some iteration:

- it violates a threshold on the training set error,
- it violates a threshold on the reciprocal condition number,
• it meets an upper bound on the number of iterations (sparsity), and/or
• it detects some minima in trends of various generalization metrics, such as the approximate LOO (see Section 3.2.3).

Stopping criteria such as these are input parameters which must be pre-defined by the user based on heuristics or based on some prior knowledge of the problem. A technique which is often used in machine learning to decide on important model parameters is \( \nu \)-fold cross validation [Bengio & Grandvalet, 2004]. By splitting the training data into \( \nu \) chunks, we may build \( \nu \) predictors, each trained on all but one unique chunk. The left-out chunk is used to evaluate a validation error for that specific predictor. The validation error is then used to rank the predictors and decide on the best model-specific parameter for a “production” predictor built using the entire training set.

For maximum efficiency during implementation, we can split the regressor matrix \( A = [\phi_1 \phi_2 \ldots \phi_d] \) into its training split rows indexed by \( G_{\text{train}} \) and validation split rows indexed by \( G_{\text{valid}} \). While training using the columns of \( A(G_{\text{train}}, \cdot) \), the chosen column indices are saved in \( I \) and, after solving for the weights \( \alpha \), the validation set residual can be calculated with a single matrix-vector product as:

\[
r_{\text{valid}} = y(G_{\text{valid}}) - A(G_{\text{valid}}, I)\alpha.
\] (5.29)

It should be noted that the use of cross validation was not proposed in our greedy RBF approximation algorithm in Chapter 3 because the selection criterion relied on RBFs which are centred at the training points. Since each of the individual predictors built using cross validation are built using only a subset of the training data, the algorithm would not have access to the same dictionary of RBFs when building each individual predictor. In the case of general dictionaries, however, each of the splits may use the entire set of basis function candidates.

We propose using \( \nu \)-fold cross validation to approximate an optimal lower bound for the training set error, \( \delta_{\text{opt}} \). By monitoring the mean squared error over the training and validation sets for each of the \( \nu \) predictors, we can construct \( \nu \) residual curves of \( \delta_{\text{train}} \) vs \( \delta_{\text{valid}} \). Finally, by averaging these curves, we can find \( \delta_{\text{opt}} \) as the training set error corresponding to the minimum validation error in the average residual curve. The individual \( \nu \) predictors may use a variety of stopping criteria. Although it is best to keep these simple, such as a maximum number of iterations, our experiments show that stopping when detecting an increase in the moving average of the validation error is quite effective. Figure 5.1 demonstrates this process for a synthetic data set of \( m = 250 \) 1-dimensional examples modelled using 3-fold cross validation. A dictionary of \( d = \ldots \)
Figure 5.1: Curves of training error vs validation error for three splits of a synthetic dataset of \( m = 250 \) examples, and the average of the curves (---). The minimum of the average curve (●) is used to define \( \delta_{\text{opt}} = 8.02E - 2 \) – a minimum bound on the training residual for a production run which builds a predictor on the full training set.

3000 Gaussian basis functions are considered. The basis functions are split into six groups of 500, where each group has a unique shape parameter ranging from \( 1E - 5 \) to \( 1E0 \). Due to the flexibility of a general dictionary, we can place basis functions at any point, so the basis function centres in each group were scattered uniformly throughout \([-0.01, 1.01]\]. The minimum average validation set error of \( 1.55E - 1 \) was found at an average training set error of \( \delta_{\text{opt}} = 8.02E - 2 \). A production run of the algorithm, using the entire training set was launched and terminated at the iteration when the training set error was below \( \delta_{\text{opt}} \). Figure 5.2 shows the approximation obtained by this algorithm. After only \( t = 16 \) iterations, \( \delta_{\text{opt}} \) was achieved and provided a predictor with excellent regression performance. This is an example of the cross-validation based greedy general dictionary learning (GGDL-CV) algorithm being able to effectively determine its own optimal predictor sparsity based only on the available data.

### 5.3.4 Parallelization

When implementing machine learning techniques for computationally-demanding tasks, we typically desire to take full advantage of modern multi-core processors and large multi-node clusters. However, the sequential nature of iterative greedy algorithms make them difficult to parallelize. We observe that the algorithm described here must be launched
Figure 5.2: GGDL-CV approximation of a synthetic dataset of \( m = 250 \) examples. 3-fold cross validation is used to obtain a minimum training set error for a production predictor built on the entire training set. The final predictor consists of 16 Gaussian basis functions with a variety of shape parameters, scattered throughout the domain.

To start, we note that the columns of \( A \) can be pre-computed before the greedy algorithm begins. This process is trivial to parallelize by splitting the basis functions among the available nodes and evaluating them at each of the \( m \) training set examples.

In addition, we can leverage parallelization in the \( \nu \)-fold cross validation process through which we obtain a good termination criterion in the form of a minimum lower bound on the training set error. By parallelizing the individual folds into “threads”, the \( \nu \) predictors can be built in parallel. The production predictor can also be constructed in parallel with the cross validation threads if we keep track of the training set residual for each iteration of the greedy algorithm. Once the cross validation threads have completed and the residual curves have been averaged, the production predictor can be modified by “backtracking” – eliminating columns from the QR factorization – until the training set residual meets \( \delta_{opt} \). Once the QR factorization has been finalized, we can solve for the weights \( \alpha \) to finalize the predictor.

Ideally, the number of cross validation folds is one less than the number of available nodes in the cluster. That way, the individual threads and the production run can be
split among the nodes. In doing so, we can ensure that the only section of the procedure where nodes are idle is the final “backtracking” modification of the production run.

5.3.5 Algorithm Block

Algorithm 5 outlines the greedy QR procedure for learning from general dictionaries using cross-validation (GGDL-CV). Ideally, the cross validation for loop used to calculate $\delta_{opt}$ should be implemented in parallel and yields a total of $\nu$ unique $\delta_{train}$ vs $\delta_{valid}$ curves, as well as $Q$ and $r$ from thread $\nu + 1$. The final weights $\alpha$ are calculated from the QR system of thread $\nu + 1$.

5.4 Randomized Algorithm for Larger Dictionaries

In Section 5.3.2 we made use of a projection matrix to significantly reduce the cost associated with ranking candidate basis functions at each iteration of our greedy OMP algorithm for learning from general dictionaries. Even with this efficient formulation, the candidate selection step may still be a numerical bottleneck in the algorithm when using very large dictionaries. Large dictionaries may be common for problems with large datasets and/or problems with a collection of diverse basis function types or shapes.

As an example, we consider a dataset of $m = 100,000$ points, where we would like to use Gaussian RBFs with varying shape parameters. We would begin by defining a basis function subset consisting of 100,000 Gaussians with a single shape parameter $\sigma$, centred on the training set. We would then duplicate this subset a number of times, each time with a different value of $\sigma$. Given as few as 10 different $\sigma$ values, the candidate selection step would require the evaluation of a future residual for $d = 1,000,000$ basis functions. If we also add multiquadric and polynomial basis functions to this dictionary, each with a range of shape parameters, the number of candidate basis functions would triple to 3,000,000.

Although our algorithm presents an opportunity to tackle this example with an incredibly flexible set of basis functions, it is clear that the candidate selection step would present a significant cost obstacle. In this section, we offer a randomized algorithm which extends our earlier OMP-based greedy procedure to handle large dictionaries. The following subsections detail the key ingredients of the proposed randomized algorithm. An algorithm block is outlined at the end of the section.
Algorithm 5 GGDL-CV: Greedy QR procedure for learning from general dictionaries using stopping criteria based on cross-validation.

Inputs:
- Training dataset $\{(x_1, y_1), \ldots, (x_m, y_m)\}$,
- Dictionary of basis functions $\{\phi_j(x)\}_{j \in D}$,
- Number of cross validation splits $\nu$,
- Stopping criteria ($t_{\text{max}}$, $\delta_{\text{min}}$, etc).

Outputs (at iteration $t$):
- Indices of chosen basis functions $I(t)$,
- Vector of weights $\alpha(t)$.

1. Compute dictionary of vectors $\{\phi_j\}_{j \in D}$ where $\phi_j = [\phi_j(x_1), \ldots, \phi_j(x_m)]^T \in \mathbb{R}^m$.
2. Compute initial denominator for each candidate $\varphi_j^{(0)} = ||\phi_j||^2$ for $j \in D^{(0)}$.
3. Split training data into $\nu$ chunks.
   for $\text{thread} = 1$ to $\nu + 1$ do
     Threads 1 to $\nu$ use all but one unique chunk to train, left out chunk to validate.
     Thread $\nu + 1$ uses full training data.
   4. $t \leftarrow 0$, $I^{(0)} \leftarrow \emptyset$, $r^{(0)} \leftarrow y$, $Q^{(0)} \leftarrow []$, $R^{(0)} \leftarrow []$.
   repeat
     5. $t \leftarrow t + 1$.
     6. Update denominators: $\varphi_j^{(t)} \leftarrow \varphi_j^{(t-1)} - \left[(Q^{(t-1)})^T \phi_j\right]^2$ for all $j \in D^{(t-1)}$.
     7. Choose best candidate: $i^{(t)} = \arg \max_{j \in D^{(t-1)}} (\phi_j^T r^{(t-1)})^2/\varphi_j$.
     8. Update set of chosen basis functions: $I^{(t)} \leftarrow I^{(t-1)} \cup i^{(t)}$.
     9. Update dictionary of candidate basis functions: $D^{(t)} \leftarrow D^{(t-1)} \setminus i^{(t)}$.
     10. Update $Q^{(t-1)}$ and $R^{(t-1)}$ to $Q^{(t)}$ and $R^{(t)}$ with $\phi_{i^{(t)}}$.
     11. Update residual: $r^{(t)} \leftarrow r^{(t-1)} - (Q^{(t)}(:,t))^T y Q^{(t)}(:,t)$.
     12. Calculate stopping criteria, including training error: $\delta_{\text{train}}^{(t)} = ||r^{(t)}||_2/\|y\|_2$.
       if $\text{thread} \in [1, \nu]$ then
         13. Solve for weights from the system: $R^{(t)} \alpha^{(t)} = (Q^{(t)})^T y$.
         14. Calculate validation set residual $r_{\text{valid}}$ and error $\delta_{\text{valid}}^{(t)}$.
       end if
     until Stopping criteria are met.
   end for

15. Average $\delta_{\text{train}}$ vs $\delta_{\text{valid}}$ curves from threads 1 to $\nu$ to find $\delta_{\text{opt}}$.
16. Backtrack: remove columns from QR factorization of thread $\nu + 1$ until $\delta_{\text{train}} \approx \delta_{\text{opt}}$.

17. Solve for weights from the system: $R^{(t)} \alpha^{(t)} = (Q^{(t)})^T y$. 


5.4.1 Random Candidate Sampling

To address the computational hurdle associated with searching over large over-specified dictionaries, we turn to randomization. Randomized algorithms have recently been used in large-scale machine learning problems with great success, most notably with stochastic gradient descent algorithms (Zhang, 2004).

We propose taking advantage of work by Smola & Schölkopf (2000) on a probabilistic speedup for approximating matrices using a greedy iterative procedure. The key result we use is stated below:

**Lemma 4.** For a uniform distribution of independent and identically distributed random variables (i.i.d) on \([0, 1]\), a random subsample of size \(\log z / \log p\) will guarantee an estimate among the best \(z \in [0, 1)\) of all candidates, with a probability \(p \in [0, 1)\).

Smola & Bartlett (2001) used this result to construct sparse greedy Gaussian process models for regression. Similarly, we make the assumption that the future residuals for candidate basis functions are i.i.d (with uniform distributions). Thus, by randomly sampling a subset \(\tilde{D}\) of about \(\log 0.05 / \log 0.95 \approx 60\) basis functions from the original candidate set \(D\), we are 95% confident of sampling a basis function in the top 5% of all candidate basis functions. Given a dictionary of \(m\) basis functions centred at the training set examples, this randomization strategy is responsible for decreasing the cost associated with the candidate selection step from \(O(mtd)\) to \(O(60m)\).

To further reduce cost, we forego precomputing the regressor matrix \(A = [\phi_j \in D]\). Instead, individual vectors can be computed at each iteration to form a local regressor matrix \(\tilde{A} = [\phi_j \in \tilde{D}]\) from which a single column can be chosen. Although this introduces additional cost at each iteration, careful bookkeeping in the implementation of this algorithm can guarantee that no column is computed more than once, making sure that computing candidate columns \(\phi\), for the randomized procedure will always be cheaper.

This randomization procedure forces us to abandon the exact expression for the future residual used to rank candidate basis functions. Since at each iteration we consider only 60 of the roughly \(d\) candidates, it would no longer be computationally efficient to continue to update \(\varphi_j^{(t)} = \| P_{K}^{(t)} \phi_j \|_2^2 \) for all \(j \in D^{(t)}\) at each iteration \(t\) (see Section 5.3.2). As an alternative, we make use of (5.22) to obtain an approximation of the future residual for the much smaller dictionary of basis function candidates.

Figure 5.3 compares training set error and test set error for the GGDL algorithm with a full candidate dictionary and 25 instances of GGDL which use a randomized sub-dictionary of only 60 basis functions. The popular Abalone dataset (see Appendix B) is used for this example, with 2000 training examples and 2177 test examples. The
dictionary consists of ten Gaussian RBFs centred at each training example, for a total of 20,000 basis functions. Accuracy results are comparable to state of the art methods such as Gaussian process models. We observe that the full dictionary strategy is more effective at driving the training residual down quickly. However, we also observe a stagnating, and eventually increasing, test set error at levels higher than the randomized strategy. In general, we often observe improved generalization when applying randomized dictionaries to real world datasets as a randomized technique is less likely to overfit the training set.

Figure 5.3: Plots of training set error (top) and test set error (bottom) for the GGDL algorithm applied to a training sample from the Abalone dataset ($m = 2000$). The dictionary is 20,000 Gaussian RBFs. One version of the algorithm uses the full dictionary of at each iteration, the other uses a randomized subdictionary of only 60/20,000 basis functions. The randomized algorithm is applied 25 times.
5.4.2 Stopping Criteria

Unlike the OMP-based greedy QR procedure from the previous section, cross-validation is not a suitable approach for determining a stopping criterion for this algorithm. The averaging of training and validation set error curves loses its accuracy as each validation instance would have a unique candidate dictionary. As a result, we resort to using the stopping criteria from the greedy RBF algorithm presented in Chapter 3. We propose that the stopping criteria for the randomized greedy QR procedure presented in this section are a combination of the following:

- A lower limit on the training set error
- A lower limit on the reciprocal condition number
- An upper limit on the number of iterations
- A check for an increase in the moving average of generalization metrics such as the leave-one-out error or minimum description length

Algorithm 6 outlines our randomized greedy procedure for learning from general dictionaries (GGDL-R).

5.5 Summary and Concluding Remarks

We began this chapter by discussing the motivation behind our greedy algorithms for general dictionary learning. There were three major motivating points which suggested that a general dictionary learning approach may offer great modelling flexibility and lead to improved results: (i) it is often difficult to choose the best kernel type to use with classical RBF approximations, (ii) although classical RBF models place basis functions at the training data, it may be desirable to place basis functions at a variety of locations within, or even outside, the problem domain, and (iii) it is desirable for an algorithm to be able to mix kernel types to capture problem-specific trends with greater flexibility – we shall demonstrate this numerically in the next chapter.

We followed the motivating points by introducing the concept of multiple kernel learning (MKL) and reviewing a recent greedy approach to MKL from Sindhwani & Lozano (2011). Having established the motivation and reviewed the literature, we next introduced a parallel algorithm for general dictionary learning (GGDL-CV). We used work from the field of sparse approximate preconditioners to rank candidate basis functions through an inexpensive expression for the future training residual and described a novel
Algorithm 6 GGDL-R: Randomized greedy QR procedure for learning from general dictionaries.

**Inputs:**
- Training dataset \( \{(x_1, y_1), \ldots, x_m, y_m\} \),
- Dictionary of basis functions \( \{\phi_j(x)\}_{j \in \mathcal{D}} \),
- Stopping criteria \( (t_{\text{max}}, \delta_{\text{min}}, \text{etc}) \).

**Outputs** (at iteration \( t \)):
- Indices of chosen basis functions \( \mathcal{I}^{(t)} \),
- Vector of weights \( \alpha^{(t)} \).

1. \( t \leftarrow 0, \mathcal{I}^{(0)} \leftarrow \emptyset, r^{(0)} \leftarrow y, Q^{(0)} \leftarrow [\ ], r^{(0)} \leftarrow [\ ] \).

repeat
2. \( t \leftarrow t + 1 \).
3. Randomly draw \( \mathcal{D} \) of size \( \approx 60 \) from \( \mathcal{D}^{(t-1)} \).
4. Compute candidate columns: \( \{\phi_j\}_{j \in \mathcal{D}} \) where \( \phi_j = [\phi_j(x_1), \ldots, \phi_j(x_m)]^T \in \mathbb{R}^m \).
5. Choose best candidate: \( i^{(t)} = \arg \max_{j \in \mathcal{D}} (\phi_j^T r^{(t-1)})^2 / \|\phi_j\|_2^2 \).
6. Update set of chosen basis functions: \( \mathcal{I}^{(t)} \leftarrow \mathcal{I}^{(t-1)} \cup i^{(t)} \).
7. Update dictionary of candidate basis functions: \( \mathcal{D}^{(t)} \leftarrow \mathcal{D}^{(t-1)} \setminus i^{(t)} \).
8. Update \( Q^{(t-1)} \) and \( R^{(t-1)} \) to \( Q^{(t)} \) and \( R^{(t)} \) with \( \phi_{i^{(t)}} \).
9. Update residual: \( r^{(t)} \leftarrow r^{(t-1)} - (Q^{(t)}(:, t))^T y Q^{(t)}(:, t) \).
10. Calculate stopping criteria.
until Stopping criteria are met.

11. Solve for weights from the system: \( R^{(t)} \alpha^{(t)} = (Q^{(t)})^T y \).

cross validation-based approach for approximating an optimal lower bound for the training set error to act as the dominant stopping criterion for this algorithm. The section concluded with additional discussion of the parallelization potential of this algorithm.

We next introduced a randomized algorithm for larger dictionaries (GGDL-R) to complement the parallel cross-validation algorithm (GGDL-CV). This algorithm was motivated by problems for which either the dataset or the dictionary of candidate basis functions is very large. GGDL-R benefits from random candidate sampling which provides a probabilistic speedup as well as an approximate expression for the future residual of each candidate. The stopping criteria for GGDL-R borrow our collection of stopping criteria from GRBF in Chapter 3, including a moving average of the approximate leave-one-out error. Algorithm 6 outlines the final step-wise procedure for GGDL-R.
Chapter 6
Numerical Results and Case Studies for GGDL

In this chapter we present a detailed set of numerical studies using the GGDL algorithm in both its parallel cross-validation form (GGDL-CV) and randomized form (GGDL-R). We begin with demonstrative examples showing the evolution of predictors for both GGDL-CV and GGDL-R as well as a number of interesting features of the GGDL algorithm: shape parameter tuning, RBF centre sampling, and learning with multiple kernels. We follow with a section of results where we compare the performance of the GGDL algorithm to commonly used machine learning algorithms for popular machine learning datasets. Finally, we end this chapter with a section dedicated to two case studies: applying the GGDL algorithm to construct sparse polynomial chaos expansions for uncertainty quantification, and to a “gappy” data problem from real-world particle image velocimetry measurements.

6.1 Demonstrative Examples

The purpose of this section is to illustrate the capabilities of GGDL by isolating unique cases where the algorithm’s features lead to improved performance. Similar to Chapter 4, we first use a test function to show the evolution of predictors for both GGDL-CV and GGDL-R. We then demonstrate three unique features of the GGDL algorithm:

1. The algorithm is capable of tuning individual basis function shape parameters through an expansion of the basis function dictionary.

2. The algorithm allows for the selection of RBF centres that are distinct from the training points, which is much more flexible than our previous GRBF algorithm,
that uses training-data centred basis functions.

3. The algorithm is capable of using more than one kernel type for approximating the training dataset.

6.1.1 GGDL-CV Predictor Evolution

We first illustrate the evolution of a predictor constructed using the cross validation-based greedy algorithm for general dictionary learning (GGDL-CV). To obtain a training dataset, we sample the $F_3$ test function (see Appendix A) 50 times using a linearly-spaced grid on $x \in [0, 1]$. After evaluating $F_3$ at each example, we apply a $\mathcal{N}(0, 16)$ noise to the outputs $y$ before normalizing them to zero mean and unit variance. Due to the training dataset’s small size, we choose a total of $k = 2$ folds for cross-validation. To get a taste of the algorithm’s flexibility, the dictionary $\mathcal{D}$ of basis functions for this approximation was chosen to be (see Table 2.2 for definitions):

(i) 25 Gaussian RBFs with $\sigma = 0.1$ and centres equally spaced on $x \in [0, 1]$

(ii) 25 multiquadric RBFs with $\sigma = 0.2$ and centres randomly sampled on $x \in [0, 1]$

(iii) 50 sigmoid functions with $a = 0.25$, $c = 0$, centred at the training data examples

Figure 6.1 shows the evolution of the GGDL-CV predictor through three approximation plots. The first plot shows only the first basis function chosen – a Gaussian RBF, with $\sigma = 0.1$, centred at $x = 1$. The second plot shows the approximation at $t = 5$ iterations. This time, five Gaussians are used and placed at $x = \{1, 0.708, 0.417, 0, 0.208\}$. The third plot shows the final approximation once the GGDL-CV algorithm has detected an increase in the moving average cross validation error.

The GGDL-CV algorithm determined that $t = 9$ iterations was the best stopping point for this problem. We observe in Figure 6.1(c) that the predictor does a very good job of approximating the true function based on the noisy dataset. Due to the GGDL algorithm’s flexibility, the final approximation included a “heterogenous” mix of basis functions. Given the smooth nature of the dataset, we are not surprised to see a predictor consisting of many Gaussians. Figure 6.2 helps visualize the type of basis functions chosen based on their centres. Unlike the GRBF algorithm developed in Chapter 3, where RBFs may not be added at the same centre more than once, the GGDL-CV approximation for this problem places both Gaussian and sigmoid basis function at $x = 1.0$.

We note here that a dictionary consisting strictly of centered basis functions is used for this example in order to best show the flexibility of the GGDL algorithm by being
Figure 6.1: A linearly-spaced dataset of 50 examples is sampled from the $F_3$ test function, perturbed with $\mathcal{N}(0, 16)$ noise, and approximated using the GGDL-CV algorithm. A dictionary of 100 basis functions across three different kernel types is used. The predictor is shown for $t = 1$ (a), $t = 5$ (b), and $t = 9$ (c) iterations.
able to illustrate the chosen basis functions via a plot (Figure 6.2). We may have also added global basis functions, such as polynomials (not to be confused with the polynomial kernel), to the dictionary. Table 6.1 outlines the basis functions chosen at each iteration of the algorithm.

Figure 6.2: The final 9-iteration predictor from Figure 6.1 is plotted with the centres of its chosen basis function. The predictor consists of seven Gaussian RBFs, one sigmoid basis function, and one multiquadric RBF.

Table 6.1: List of kernel types, RBF centres, and shape parameters for the final predictor from Figure 6.1. A total of nine basis functions are chosen and vary between three different kernel types. Two different basis functions are placed at $x = 1.000$.

<table>
<thead>
<tr>
<th>iteration</th>
<th>kernel type</th>
<th>RBF centre</th>
<th>shape parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Gaussian</td>
<td>$x = 1.000$</td>
<td>$\sigma = 0.1$</td>
</tr>
<tr>
<td>2</td>
<td>Gaussian</td>
<td>$x = 0.708$</td>
<td>$\sigma = 0.1$</td>
</tr>
<tr>
<td>3</td>
<td>Gaussian</td>
<td>$x = 0.417$</td>
<td>$\sigma = 0.1$</td>
</tr>
<tr>
<td>4</td>
<td>Gaussian</td>
<td>$x = 0.000$</td>
<td>$\sigma = 0.1$</td>
</tr>
<tr>
<td>5</td>
<td>Gaussian</td>
<td>$x = 0.208$</td>
<td>$\sigma = 0.1$</td>
</tr>
<tr>
<td>6</td>
<td>Sigmoid</td>
<td>$x = 1.000$</td>
<td>$a = 0.25, \ c = 0$</td>
</tr>
<tr>
<td>7</td>
<td>Multiquadric</td>
<td>$x = 0.382$</td>
<td>$\sigma = 0.2$</td>
</tr>
<tr>
<td>8</td>
<td>Gaussian</td>
<td>$x = 0.833$</td>
<td>$\sigma = 0.1$</td>
</tr>
<tr>
<td>9</td>
<td>Gaussian</td>
<td>$x = 0.542$</td>
<td>$\sigma = 0.1$</td>
</tr>
</tbody>
</table>
6.1.2 GGDL-R Predictor Evolution

We now use Figure 6.3 to demonstrate the evolution of a predictor constructed using the randomized greedy algorithm for general dictionary learning (GGDL-R). We consider the same training dataset used to illustrate the evolution of a GGDL-CV approximation in Figure 6.1: 50 linearly-spaced points on $x \in [0, 1]$, evaluated using the $F_3$ test function and perturbed with $\mathcal{N}(0, 16)$ noise. We once again note that after applying noise, the output values $y$ are normalized to zero mean and unit variance.

To approximate using GGDL-R, we use the same dictionary of 100 basis functions: 25 Gaussian RBFs ($\sigma = 0.1$) spaced equally on $x = [0, 1]$, 25 multiquadric RBFs ($\sigma = 0.2$) spaced randomly on $x = [0, 1]$, and 50 sigmoid functions ($a = 0.25$, $c = 0$) centred at the training examples. Each of the three instances of GGDL-R uses a randomized subdictionary of only five of the 100 basis functions from $\mathcal{D}$ at each iteration. We observe a clear variance in the predictors during the early stages of the algorithm: $t = 1$, and $t = 5$. The three instances of GGDL-R all finally stopped upon detecting an increase in the moving average of LOO after $t = 11$, $t = 10$, and $t = 10$ iterations. The three final predictors show very little variance in Figure 6.3(c). In fact, the three predictors had final test set errors (evaluated using a 200-point sample of $F_3$ on $x \in [0, 1]$) which varied by only 0.7%. The test set errors were 0.12423, 0.12337, and 0.12393.

In Table 6.2 we list the various basis functions chosen at each iteration of the three GGDL-R trials. Due to the small subdictionary size, we observe a great variety of basis functions between the three trials. Of the eleven chosen basis functions in trial 1, three were Gaussians. In trial 2, three of ten were Gaussians.

In addition to the small variance between the three final GGDL-R approximations, we observe little difference between the GGDL-R predictors and the GGDL-CV predictor constructed in the previous subsection. Figure 6.4 plots all four approximations. Compared to the GGDL-R predictors, the GGDL-CV predictor had a lower test set error (0.12057), and used one or two fewer basis functions. It was however more costly to construct not only because of the cross-validation stage of evaluating the error on a validation set for each thread, but also because the GGDL-CV procedure required a full computation of the regressor matrix $A$, which in this case had 50 rows and 100 columns (one for each basis function). The GGDL-R procedure only needed to compute 5 columns at each iteration, which resulted in a total of at most 80 columns for the $t = 11$ approximation which backtracked from $t = 16$ upon detecting an increase in LOO.

The tradeoff between cost and accuracy is one of the most common problems in machine learning and is something we observe in this example. Nonetheless, considering the noisy dataset at hand, both GGDL-CV and GGDL-R provide excellent approximations.
Figure 6.3: A linearly-spaced dataset of 50 examples is sampled from the $F_3$ test function, perturbed with $\mathcal{N}(0,16)$ noise, and approximated three times using the GGDL-R algorithm. A dictionary of 100 basis functions across three different kernel types is used. The predictors are shown at $t = 1$, $t = 5$, and the final iterations ((a),(b), and (c) respectively).
Table 6.2: List of kernel types, RBF centres, and shape parameters for the final predictors from Figure 6.3. The first GGDL-R trial used a total of 11 basis functions, while both the second and third trial used a total of 10. Each of the three trials stopped after detecting an increase in the moving average of LOO. We intentionally specified a small subdictionary sample size of only 5 of the total 100 basis functions in $D$. The result is predictors with a lot of variance in their composition.

<table>
<thead>
<tr>
<th>iteration</th>
<th>kernel type</th>
<th>GGDL-R Trial 1</th>
<th>kernel type</th>
<th>GGDL-R Trial 2</th>
<th>kernel type</th>
<th>GGDL-R Trial 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RBF centre</td>
<td>shape parameters</td>
<td>RBF centre</td>
<td>shape parameters</td>
<td>RBF centre</td>
</tr>
<tr>
<td>1</td>
<td>Sigmoid</td>
<td>$x = 0.816$</td>
<td>$a = 0.25$, $c = 0$</td>
<td>Sigmoid</td>
<td>$x = 0.102$</td>
<td>$a = 0.25$, $c = 0$</td>
</tr>
<tr>
<td>2</td>
<td>Gaussian</td>
<td>$x = 0.333$</td>
<td>$\sigma = 0.1$</td>
<td>Multiquadric</td>
<td>$x = 0.772$</td>
<td>$\sigma = 0.2$</td>
</tr>
<tr>
<td>3</td>
<td>Sigmoid</td>
<td>$x = 0.714$</td>
<td>$a = 0.25$, $c = 0$</td>
<td>Sigmoid</td>
<td>$x = 0.245$</td>
<td>$a = 0.25$, $c = 0$</td>
</tr>
<tr>
<td>4</td>
<td>Multiquadric</td>
<td>$x = 0.698$</td>
<td>$\sigma = 0.2$</td>
<td>Gaussian</td>
<td>$x = 0.875$</td>
<td>$\sigma = 0.1$</td>
</tr>
<tr>
<td>5</td>
<td>Sigmoid</td>
<td>$x = 0.898$</td>
<td>$a = 0.25$, $c = 0$</td>
<td>Sigmoid</td>
<td>$x = 0.531$</td>
<td>$a = 0.25$, $c = 0$</td>
</tr>
<tr>
<td>6</td>
<td>Multiquadric</td>
<td>$x = 0.772$</td>
<td>$\sigma = 0.2$</td>
<td>Multiquadric</td>
<td>$x = 0.549$</td>
<td>$\sigma = 0.2$</td>
</tr>
<tr>
<td>7</td>
<td>Gaussian</td>
<td>$x = 0.458$</td>
<td>$\sigma = 0.1$</td>
<td>Sigmoid</td>
<td>$x = 0.939$</td>
<td>$a = 0.25$, $c = 0$</td>
</tr>
<tr>
<td>8</td>
<td>Sigmoid</td>
<td>$x = 0.367$</td>
<td>$a = 0.25$, $c = 0$</td>
<td>Gaussian</td>
<td>$x = 0.375$</td>
<td>$\sigma = 0.1$</td>
</tr>
<tr>
<td>9</td>
<td>Gaussian</td>
<td>$x = 0.500$</td>
<td>$\sigma = 0.1$</td>
<td>Sigmoid</td>
<td>$x = 0.633$</td>
<td>$a = 0.25$, $c = 0$</td>
</tr>
<tr>
<td>10</td>
<td>Sigmoid</td>
<td>$x = 0.204$</td>
<td>$a = 0.25$, $c = 0$</td>
<td>Gaussian</td>
<td>$x = 0.625$</td>
<td>$\sigma = 0.1$</td>
</tr>
<tr>
<td>11</td>
<td>Multiquadric</td>
<td>$x = 0.618$</td>
<td>$\sigma = 0.2$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Figure 6.4: The final 9-iteration GGDL-CV predictor from Figure 6.1 is compared to the three final GGDL-R predictors from Figure 6.3. Based on a test set of 200 examples sampled from the true function $F_3$, the GGDL-CV predictor has a test set error of 0.12057 while the $t = 11$, $t = 10$, and $t = 10$ GGDL-R predictors have test set errors of 0.12423, 0.12337, and 0.12393, respectively.

6.1.3 Shape Parameter Tuning

Similar to our GRBF algorithm from Chapter 3, the GGDL algorithm may be used to tune shape parameters for basis functions of a pre-determined kernel type. At each iteration, the GRBF algorithm first chooses an RBF centre by sorting through the residual. It may then utilize a variety of optimization algorithms to find the shape parameter value which best reduces the training set residual. The GGDL algorithm performs in a similar way but is less “naive” in its choice of RBF centre. By defining a large dictionary of RBFs with a variety of shape parameters at each centre, the GGDL algorithm will choose the basis function (combination of centre and shape parameter) which best reduces the training residual.

We should note here that although in our implementation of GRBF we have found that a simple grid search for the shape parameters performs quite well, the GRBF technique allows the user to instead use optimization schemes with more resolution. GGDL, on the other hand, requires a discretized dictionary of basis functions to act as the candidate set at each iteration of the greedy algorithm.

To compare GRBF and GGDL for shape parameter tuning, we sample 50 linearly-spaced training examples on $x \in [0, 1]$, evaluate the one-dimensional variant of the $F_7$ test function, and apply a $\mathcal{N}(0, 10^4)$ noise before normalizing the outputs to zero mean and unit variance.
When tackling this approximation problem with GRBF, we use Gaussian RBFs and explicitly specify a shape parameter (\(\sigma\)) grid of ten values between 0.001 and 1 to search through for each new RBF addition. For a fair comparison, we tackle the approximation problem with GGDL in a nearly identical fashion by defining a dictionary of basis functions which are all Gaussians, centred at the training examples and have the same shape parameter values. As a result, the regressor matrix \(A\) in this case has 50 rows and 500 columns. In addition, the GGDL algorithm used here is neither the GGDL-CV nor the GGDL-R version. For a fair comparison with GRBF, we use a single GGDL “thread” (with no cross-validation) and rely on the moving average of LOO as the dominant stopping criteria for both.

Figure 6.5 illustrates the predictors obtained from the two methods. There is little observable difference between the two, but it is noteworthy that the GRBF algorithm executed 6 additional iterations before detecting an increase in the moving average LOO. The final GRBF predictor had a test set error of 0.314, while the GGDL predictor had a test set error of 0.300.

![Figure 6.5](image)

Figure 6.5: A noisy 50-point dataset from the one-dimensional variant of the \(F_7\) test function is approximated using GRBF and GGDL. Both algorithms use Gaussian RBFs centred at the training examples and the same candidate shape parameter values. GRBF constructs an approximation with 15 basis functions, GGDL constructs one with 9.

Table 6.3 lists the RBFs chosen at each iteration by the two algorithms. We should also note here that, unlike GRBF, the GGDL procedure allows us to place more than one basis function at a single centre, as long as the two have different shape parameters. In fact, iterations 4 and 9 of the GGDL approximation for this problem placed Gaussians at \(x = 0.796\), once with \(\sigma = 0.001\) and once with \(\sigma = 0.002\). This allows for an additional level of modelling flexibility that GRBF does not have. GRBF eliminates an RBF centre
from the greedy candidate set after using it once. GGDL on the other hand eliminates only a single basis function from the dictionary $\mathcal{D}$.

Table 6.3: List of kernel types, RBF centres, and shape parameters for the final predictors from Figure 6.5. Both algorithms stop after detecting an increase in the moving average of LOO, with GRBF stopping after 15 iterations and GGDL stopping after 9 iterations.

<table>
<thead>
<tr>
<th>iteration</th>
<th>GRBF</th>
<th>GGDL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RBF centre</td>
<td>shape parameter</td>
</tr>
<tr>
<td>1</td>
<td>$x = 0.061$</td>
<td>$\sigma = 0.002$</td>
</tr>
<tr>
<td>2</td>
<td>$x = 0.184$</td>
<td>$\sigma = 0.001$</td>
</tr>
<tr>
<td>3</td>
<td>$x = 0.939$</td>
<td>$\sigma = 0.001$</td>
</tr>
<tr>
<td>4</td>
<td>$x = 0.796$</td>
<td>$\sigma = 0.001$</td>
</tr>
<tr>
<td>5</td>
<td>$x = 0.286$</td>
<td>$\sigma = 0.001$</td>
</tr>
<tr>
<td>6</td>
<td>$x = 0.714$</td>
<td>$\sigma = 0.001$</td>
</tr>
<tr>
<td>7</td>
<td>$x = 0.020$</td>
<td>$\sigma = 0.001$</td>
</tr>
<tr>
<td>8</td>
<td>$x = 0.878$</td>
<td>$\sigma = 0.001$</td>
</tr>
<tr>
<td>9</td>
<td>$x = 0.837$</td>
<td>$\sigma = 0.020$</td>
</tr>
<tr>
<td>10</td>
<td>$x = 0.490$</td>
<td>$\sigma = 0.100$</td>
</tr>
<tr>
<td>11</td>
<td>$x = 0.531$</td>
<td>$\sigma = 1.000$</td>
</tr>
<tr>
<td>12</td>
<td>$x = 0.265$</td>
<td>$\sigma = 0.002$</td>
</tr>
<tr>
<td>13</td>
<td>$x = 0.347$</td>
<td>$\sigma = 0.001$</td>
</tr>
<tr>
<td>14</td>
<td>$x = 0.327$</td>
<td>$\sigma = 0.002$</td>
</tr>
<tr>
<td>15</td>
<td>$x = 0.816$</td>
<td>$\sigma = 0.005$</td>
</tr>
</tbody>
</table>

More interesting, however, are the training and test error trends produced by both algorithms for this problem. Because the GGDL algorithm implements a basis function selection criterion which minimizes the exact future training residual, we expect GGDL to have the lower training set error throughout the modelling stages. Figure 6.6 illustrates the training set error trends for both algorithms. The GGDL approximation does in fact drive the residual down faster. This can be attributed to its exact future residual-based candidate selection criteria. Figure 6.7 illustrates the test set error trends. GGDL once again performs better, reaching a lower test set error in fewer iterations. It is worth nothing that the GRBF algorithm could also have reached a fairly low test set error and both algorithms appear to have been a little handicapped with an “early stop” via their LOO trends. The ideal stopping point for GRBF would have been around $t = 21$, and for GGDL around $t = 13$.

Although it may appear that, due to its more “accurate” selection criteria, GGDL is the better algorithm for this case, we must not forget the difference in computational cost. GGDL begins with a complete computation of the 500-column regressor matrix.
A. GRBF, on the other hand, computes only 10 columns at each iteration (one for each candidate shape parameter value) for a total of 200 columns at \( t = 20 \) before backtracking to \( t = 15 \). At the same time, GGDL requires has high “upfront” memory cost associated with storing the full regressor matrix. Depending on the modelling requirements, the additional computational and memory costs associated with training a GGDL predictor may outweigh the accuracy and sparsity benefits.

Figure 6.6: Training set error trends for the GRBF and GGDL approximations in Figure 6.5. The GRBF algorithm stops at iteration 15, while the GGDL algorithm stops at iteration 9. Thanks to a basis function selection criterion based on the exact future residual, GGDL is best able to drive the training error down faster.

Figure 6.7: Test set error trends for the GRBF and GGDL approximations in Figure 6.5. The GRBF algorithm stops at iteration 15, while the GGDL algorithm stops at iteration 9. Both algorithms appear to have stopped earlier than their optimal point by detecting an increase in the moving average of LOO.
6.1.4 Structurally-Sampled RBF Centres

One of the unique features of the GGDL algorithm is that, unlike GRBF, RBFs (or similar basis functions with “centres”) in the dictionary of possible predictor elements may be placed at arbitrary locations in the domain. Before conducting any experiments, we hypothesize that this may benefit the case of approximating highly unstructured datasets, such as random samples within $A'$. In addition, research has shown that some approximation problems may benefit from the placement of RBF centres outside the domain (Fornberg & Zuev, 2007) – something we discussed earlier in Chapter 3.

Once again we use the one-dimensional variant of the $F_7$ test function (Appendix A). We randomly sample 50 training examples using a uniform distribution on $x \in [0,1]$ and after evaluating the test function, apply a $\mathcal{N}(0,400)$ noise to $y$. As usual, we also normalize $y$ to zero mean and unit variance. To observe the effects of both structurally-sampled RBF centres and placing centres outside the domain, we attempt to approximate this dataset with three trials of the GGDL algorithm, each with a different set of basis function centres ($C_1$, $C_2$, $C_3$):

$C_1$: Centres at the 50 training examples

$C_2$: $C_1 + 100$ linearly spaced centres on $x \in [0,1]$

$C_3$: $C_2 + 100$ linearly spaced centres on $x \in [-0.1,0] \cup [1,1.1]$

To compose the dictionaries for each of these GGDL trials, we place ten Gaussian RBFs, each with a different shape parameter on $\sigma \in [0.001,1]$ at the listed centre sets. This results in regressor matrix sizes of 500 columns, 1500 columns, and 2500 columns for trials 1, 2, and 3, respectively. To be sure our observations are not attributed to the randomization associated with dividing the training set into cross-validation splits for GGDL-CV or to the randomization associated with sampling from the dictionary of basis functions in GGDL-R, we implement a single “thread” of GGDL here and rely on LOO as the dominant stopping criterion.

Figure 6.8 shows the three predictors constructed with the above GGDL trials. Although not immediately visible, the GGDL trial which utilizes centre set $C_3$ is best able to approximate the true function. In numerical order, the three GGDL predictors had the following test set errors: 0.352, 0.229, and 0.191. The predictors used a total of 16, 20, and 20 basis functions, respectively. It appears that the 100 structurally-sampled RBF centres added in centre set $C_2$ provided the additional basis function candidates necessary to better approximate this randomly-sampled dataset. Of the 20 basis function centres
chosen, only 3 came from the original $C_1$ set. In addition, the 100 centres outside of the
$[0,1]$ domain added in centre set $C_3$ helped better approximate the areas of the function
closer to the domain boundaries. Of the 20 centres chosen in the third GGDL trial, 2
were outside the domain.

Figures 6.9 and 6.10 show the training and test set error trends for all three GGDL
trials. Using an additional structurally-sampled set of centres, as in trials 2 and 3, appears
to have an immediate benefit in driving down the training set residual, as both of these
approximations continually outperformed the first trial. Of the three centre sets, $C_3$ was
able to yield the lowest overall test set error.

It is worth noting, however, that the results presented above are for a single ex-
ample from this test function. By conducting these experiments repeatedly, we were
not confident enough in the benefits associated with the expansion of dictionaries with
structurally-sampled centres, or those associated with including centres outside the train-
ing set domain. Improvements in accuracy and/or sparsity were observed in a majority
of the experiments conducted on test functions, like the one above. In a minority of
cases, the expansion of the candidate centre set from $C_1$ to $C_2$ or $C_3$ resulted in equal or
marginally worse results. As is typical in dictionary learning, the ultimate results are
highly dependent on the “engineering” of the candidate dictionary for the problem at
hand. Nonetheless, the above example and results are absolutely necessary to document
for future work in this area.

6.1.5 Learning with Multiple Kernel Types

One of the desired features with which we motivate the development of the GGDL algo-
thesis in Chapter 5 was constructing approximations by mixing kernel types. As discussed
in that chapter, the field of multiple kernel learning (MKL) has a number of methods
for training models which consist of more than one kernel. However, one of the unique
features of the GGDL algorithm is that, unlike other MKL algorithms, we do not need
to add the “full” kernel (basis functions at each of the $m$ training examples). Instead of
constructing a dictionary of kernels and constructing predictors from some multiple of
$m$ basis functions, GGDL uses a dictionary of basis functions and at each iteration picks
only those which are most helpful in approximating the training set.

In this subsection we will demonstrate this “kernel mixing” feature by using GGDL
to construct approximations with a Gaussian kernel, a multiquadric (MQ) kernel, and
a mixture of both. The training dataset for this example is composed of 100 randomly-
sampled points on $x \in [0,1]$. The outputs $y$ are obtained by evaluating the $F_1$ test
Figure 6.8: A randomly-sampled dataset of 50 examples from the $F_7$ test function, disturbed with $\mathcal{N}(0, 400)$ noise, and approximated three times using the GGDL algorithm. The first approximation (a) uses only training data-centred RBFs. The second approximation (b) adds 100 linearly-spaced RBFs on $x \in [0, 1]$. The third approximation (c) adds 100 centres outside the domain. The chosen RBF centres are shown as $x$. 
Figure 6.9: Training set error trends for the GGDL approximations in Figure 6.8. Expanding the basis function dictionary with structurally-sampled RBFs appears to provide a significant benefit the decreasing the training set residual with fewer iterations.

Figure 6.10: Test set error trends for the GGDL approximations in Figure 6.8. Of the three GGDL trials, the one with with RBF centres structurally sampled both inside and outside the domain appears ($C_3$) to perform best.

function (see Appendix A), applying a $\mathcal{N}(0, 0.01)$ noise, and normalizing to zero mean and unit variance. Given that we have a relatively large dataset, we use the GGDL-CV procedure here with 3-fold cross validation. The basis function dictionary for the Gaussian trial is composed of 10 Gaussian basis functions at each of the 100 training examples, each with a shape parameter on $\sigma \in [0.001, 1]$. In classical multiple kernel learning terms, we are using ten total Gaussian kernels: one with $\sigma = 0.001$, one with $\sigma = 0.002$, etc. Similarly, the MQ GGDL trial uses a dictionary of 10 MQ basis functions at each training example. The third “mixed kernel” trial simply merges the two and uses
a dictionary of 2000 basis functions (10 of each kernel type at each training example).

As is typically the case in practical applications, it is not immediately clear which kernel is the better choice for this dataset. Given the choice of the Gaussian or the multiquadric kernel, we cannot immediately assume that one will generalize better to unseen examples than the other. We hypothesize that, given a GGDL basis function dictionary composed of both kernels, the approximation constructed should be as good, if not better, than the better of the two individual kernels.

Figure 6.11 illustrates the approximations constructed using the three GGDL trials. Using the Gaussian kernel yielded an approximation with 11 basis functions and a test set error of 0.224. Using the MQ kernel yielded an approximation with 18 basis functions and a test set error of 0.237. Finally, mixing the two kernel types in our third GGDL-CV trial resulted in an approximation with a test set error of 0.211 and 11 total basis functions (10 Gaussians and 1 multiquadric).

Figures 6.12 and 6.13 show the training and test set error trends for the three GGDL trials used in this problem. The mixed kernel predictor behaves almost identically to the Gaussian kernel predictor for the first 10 iterations of the trial. It also manages to reach the lowest overall test set error. It is evident that this approximation example benefits from using a dictionary with both Gaussian and MQ basis functions. However, once again we should note that this is not guaranteed to be the case for all approximation problems. Our studies showed that many synthetic problems such as this one could benefit (in accuracy and/or sparsity) from using a dictionary with a variety of kernel types.

6.2 Machine Learning Datasets

In this section, we compare the GGDL algorithm to two common machine learning regression techniques: Gaussian process models (GPM) (Rasmussen & Williams [2006]) and support vector machines (SVMs) (Vapnik [1998]). To build GPM models, we use an efficient in-house Fortran code. To build SVM models, we use the Scikit-learn (Pedregosa et al. [2011]) machine learning distribution. Under the hood, Scikit-learn uses the popular and highly efficient Liblinear solver (Fan et al. [2008]). Using Scikit-learn allows us to take full advantage of Python integration in our codes for scripted experiments. Finally, we note that the GGDL algorithm is used in both its cross-validation form (GGDL-CV) and its randomized form (GGDL-R), where we report the averaged results after 10 trials of the GGDL-R algorithm. All experiments are conducted on the same machine: a mid 2012 MacBook Pro Retina with 16GB of memory and a 2.3 GHz Intel Core i7 processor.
Figure 6.11: A randomly-sampled dataset of 100 examples from the $F_1$ test function, disturbed with $\mathcal{N}(0,0.01)$ noise, and approximated three times using the GGDL algorithm. The first approximation (a) uses the Gaussian kernel. The second approximation (b) uses the multiquadric kernel. The third approximation (c) uses both kernels.
Figure 6.12: Training set error trends for the GGDL-CV approximations in Figure 6.11. The mixed kernel approximation is the best performer until about iteration 15, at which point the strictly Gaussian kernel is best able to drive the training error down.

Figure 6.13: Test set error trends for the GGDL-CV approximations in Figure 6.11. Given some fixed sparsity, the mixed kernel approximation is usually never the worst performer, but does have the lowest overall test error.

We do our best to simulate a practical surrogate modelling case where we must approximate a dataset for which we have no “inside knowledge” – we do not know which kernel types or shape parameters may perform favourably. The problem statements for both GPM and SVM are convex only for positive definite Mercer kernels [Vapnik 1995]. Although our algorithm is more flexible and contains no such constraint, we use the Gaussian kernel for each of the approximation methods in this example to avoid discrepancies. Because the GPM algorithm tunes the global shape parameter, no further inputs are necessary. SVM and GGDL, however, rely on a shape parameter value or a
pre-specified dictionary of basis functions, respectively. For this reason, we use three SVM approximations with $\sigma \in \{0.1, 1, 10\}$. For each of the candidate shape parameter values, we perform 3-fold cross validation and use the average error to select a single $\sigma$ to use when approximating the full training set. We report the results of this approximation on the test set. When facing the problem of modelling a new dataset, it is common practice to construct a collection of predictors such as this. For GGDL, we provide a dictionary composed of the same three Gaussian basis functions ($\sigma \in \{0.1, 1, 10\}$) centred at each of the data points. GGDL-CV uses $k = 3$ fold cross validation, while GGDL-R relies on the LOO stopping criteria as a measure of generalization.

We focus on four popular machine learning datasets, each of which is described in more detail in Appendix B. These are the Abalone, the Boston, the CPU Small, and Diabetes datasets. For each of the datasets, we average results over 20 random training/test set splits of the original dataset. For each dataset, the ratio of training and test sets are held constant. For example, the Abalone dataset, which consists of 4177 examples, is approximated with a ratio of 0.5 (2088 training points, 2089 test points). Although in each of the 20 splits the datasets are sampled randomly, the size of the problem will always remain the same. As is common practice, we normalize all input dimensions of $x$ and outputs $y$ in the training sets to zero mean and unit variance.

### 6.2.1 Abalone

The Abalone dataset is composed of $m = 4177$ examples, each with dimensionality $n = 8$. We use a training-test set ratio of 0.5 for training and test set sizes of 2088 and 2089, respectively. Figure 6.14 illustrates the test set errors for each approximation type.

The randomized GGDL algorithm proved to be the most accurate algorithm for this problem. GGDL-CV also performed quite well, but was not as accurate as GPM. Of the four algorithms, SVM appeared to be least accurate over the 20 random samples. The four algorithms had the following average training times: SVM 1.08s, GPM 101.67s, GGDL-CV 3.15s, and GGDL-R 1.07s. Finally, the algorithms produced approximations with the following average number of basis functions used: 1741.65, 2088.00, 24.20, and 47.14. As expected, both SVM and GPM produce predictors with many basis functions, while the GGDL algorithm produces very sparse predictors which are cheap-to-evaluate. At the same time, GGDL is very fast to train for this problem, and just as, if not more, accurate than the other state of the art methods considered here.
6.2.2 Boston

The Boston housing dataset is composed of \( m = 506 \) examples, each with dimensionality \( n = 13 \). It is one of machine learning’s more difficult-to-approximate datasets. We use training and test set sizes of 400 and 106, respectively. Figure 6.15 illustrates the test set errors for each approximation type using a box plot.

The GGDL algorithm does not perform well on this dataset. Both GGDL-CV and GGDL-R built predictors which are less accurate than SVM and GPM. It is also worth nothing that GGDL-R constructs a single outlying predictor with a very high test set error of over 1.4. The four algorithms had the following average training times: SVM 0.056s, GPM 7.090s, GGDL-CV 0.094s, and GGDL-R 0.118s. Finally, the algorithms produced approximations with the following average number of basis functions used: 267.8, 400.0, 5.7, and 18.8. Although GGDL is able to construct predictors with fewer basis functions than SVM or GPM, it loses out to both in accuracy. Our study finds SVM to be the best algorithm for this particular dataset.

![Figure 6.14: The GGDL-CV and GGDL-R algorithms are compared to the popular SVM and GPM regression techniques over 20 random samples of the Abalone dataset with a training set of size 2088 and test set of size 2089. The GGDL-R algorithm is applied 10 times for each of the 20 random dataset samples. This box plot shows the distribution of each algorithm’s test set error.](image)

6.2.3 CPU Small

The CPU Small dataset is composed of \( m = 8192 \) examples, each with dimensionality \( n = 12 \). We use training and test set sizes of 1000 and 7192, respectively. Figure 6.16 illustrates the test set errors for each approximation type using a box plot.

GGDL-R is the big winner for this dataset. It significantly out-performed SVM, GPM,
and even GGDL-CV models in terms of accuracy. The randomized algorithm produced models which were an order of magnitude more sparse than SVM, in the same amount of training time. The four algorithms had the following average training times: SVM 0.534s, GPM 40.319s, GGDL-CV 0.604s, and GGDL-R 0.555s. Finally, the algorithms produced approximations with the following average number of basis functions used: 542.85, 1000, 11.55, and 45.395. It is also worth noting that, apart from a single outlier, GGDL-CV approximations were more accurate than both SVM and GPM, while being the most sparse of all four predictor types.

6.2.4 Diabetes

The Diabetes dataset is composed of $m = 442$ examples, each with dimensionality $n = 10$. We use training and test set sizes of 300 and 142, respectively. Figure 6.17 illustrates the test set errors for each approximation type using a box plot.

The GGDL algorithm performs particularly well for this dataset. It compares to the SVM algorithm in terms of accuracy, with GGDL-CV producing predictors which are slightly more accurate and GGDL-R slightly less so. The four algorithms had the following average training times: SVM 0.036s, GPM 2.619s, GGDL-CV 0.048s, and GGDL-R 0.056s. Finally, the algorithms produced approximations with the following average number of basis functions used: 264.3, 300.0, 3.7, and 12.5. We conclude that, for this dataset, GGDL performs exceptionally well as it produces predictors with state-of-the-art accuracy, reasonable training cost, and fractions of the sparsity of SVM or GPM.
Figure 6.16: The GGDL-CV and GGDL-R algorithms are compared to the popular SVM and GPM regression techniques over 20 random samples of the CPU Small dataset with a training set of size 1000 and test set of size 7192. The GGDL-R algorithm is applied 10 times for each of the 20 random dataset samples. This box plot shows the distribution of each algorithm’s test set error.

6.2.5 Final Results

The following three tables document the results of the above subsections. Table 6.4 lists the average test set errors of the four approximation methods for each dataset, Table 6.5 lists the average training times, and Table 6.6 lists the average sparsity (number of basis functions used in the final predictor). Results for GGDL are shown in bold if they outperform both SVM and GPM.

Table 6.4: Average test set errors for SVM, GPM, GGDL-CV, and GGDL-R after approximating four popular machine learning datasets. 20 random splits from each dataset are approximated. GGDL-R results are averaged over 10 trials for each split. Bold result denotes GGDL outperforming SVM and GPM.

<table>
<thead>
<tr>
<th>dataset</th>
<th>SVM</th>
<th>GPM</th>
<th>GGDL-CV</th>
<th>GGDL-R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abalone</td>
<td>0.677</td>
<td>0.686</td>
<td>0.698</td>
<td>0.673</td>
</tr>
<tr>
<td>Boston</td>
<td>0.428</td>
<td>0.467</td>
<td>0.625</td>
<td>0.571</td>
</tr>
<tr>
<td>CPU Small</td>
<td>0.219</td>
<td>0.757</td>
<td>0.338</td>
<td>0.195</td>
</tr>
<tr>
<td>Diabetes</td>
<td>0.761</td>
<td>0.904</td>
<td>0.758</td>
<td>0.770</td>
</tr>
</tbody>
</table>

Apart from the Boston dataset, GGDL appears to perform as well as, and sometimes better than, state-of-the-art algorithms such as SVM and GPM, both in terms of accuracy and training time. In addition, it is always able to produce predictors with fractions of the total number of basis functions. GGDL-CV continuously produces the most sparse predictors, using a total number of basis functions which is about 1% of the training set.
Figure 6.17: The GGDL-CV and GGDL-R algorithms are compared to the popular SVM and GPM regression techniques over 20 random samples of the Diabetes dataset with a training set of size 300 and test set of size 142. The GGDL-R algorithm is applied 10 times for each of the 20 random dataset samples. This box plot shows the distribution of each algorithm’s test set error.

Table 6.5: Average training times (in seconds) for SVM, GPM, GGDL-CV, and GGDL-R after approximating four popular machine learning datasets. 20 random splits from each dataset are approximated. GGDL-R results are averaged over 10 trials for each split. Bold result denotes GGDL outperforming SVM and GPM.

<table>
<thead>
<tr>
<th>dataset</th>
<th>SVM</th>
<th>GPM</th>
<th>GGDL-CV</th>
<th>GGDL-R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abalone</td>
<td>1.802</td>
<td>103.3</td>
<td>3.013</td>
<td>1.037</td>
</tr>
<tr>
<td>Boston</td>
<td>0.100</td>
<td>6.920</td>
<td>0.093</td>
<td>0.139</td>
</tr>
<tr>
<td>CPU Small</td>
<td>0.550</td>
<td>42.95</td>
<td>0.635</td>
<td>0.555</td>
</tr>
<tr>
<td>Diabetes</td>
<td>0.063</td>
<td>2.662</td>
<td>0.048</td>
<td>0.056</td>
</tr>
</tbody>
</table>

Table 6.6: Average final predictor sparsities for SVM, GPM, GGDL-CV, and GGDL-R after approximating four popular machine learning datasets. 20 random splits from each dataset are approximated. GGDL-R results are averaged over 10 trials for each split. Bold result denotes GGDL outperforming SVM and GPM.

<table>
<thead>
<tr>
<th>dataset</th>
<th>SVM</th>
<th>GPM</th>
<th>GGDL-CV</th>
<th>GGDL-R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abalone</td>
<td>1720.6</td>
<td>2088.0</td>
<td>22.5</td>
<td>46.0</td>
</tr>
<tr>
<td>Boston</td>
<td>264.4</td>
<td>400.0</td>
<td>5.9</td>
<td>23.3</td>
</tr>
<tr>
<td>CPU Small</td>
<td>505.6</td>
<td>1000.0</td>
<td>10.2</td>
<td>43.4</td>
</tr>
<tr>
<td>Diabetes</td>
<td>262.8</td>
<td>300.0</td>
<td>4.4</td>
<td>11.4</td>
</tr>
</tbody>
</table>

size. GGDL-R is able to produce predictors with 3-4 times more basis functions, but still a fraction of the number used by SVM or GPM. For this reason, GGDL is an excellent algorithm choice when constructing a very sparse predictor is a priority. This may be
Chapter 6. Numerical Results and Case Studies for GGDL

6.3 Case Studies

6.3.1 Sparse Polynomial Chaos Expansions for Uncertainty Quantification

Uncertainty quantification (UQ) is a field of study which can benefit substantially from algorithms that build accurate and sparse approximations. As engineering simulations become more complex, it is important to have an ability to quantify the effects of variations in input data and model parameters on outputs. In UQ, we typically use a probabilistic framework where uncertain inputs are random quantities and the quantity of interest (QOI) is typically a result of the simulation. The goal of UQ is to quantify the impact of uncertainty in the inputs on the outputs.

For an \( n \)-dimensional problem, we denote the random variable inputs as \( \theta = [\theta_1, \ldots, \theta_n]^T \) and the QOI as \( u(\theta) \). Perhaps the most popular UQ technique is Monte Carlo (MC) simulation (Hastings, 1970). MC is a sampling-based method through which the statistical moments of \( u \) may be computed trivially from a collection of \( m \) samples of \( u(\theta) \). However, MC suffers from a slow rate of convergence of \( \mathcal{O}\left(\frac{1}{\sqrt{m}}\right) \). Other methods, such as the Quasi-Monte Carlo (QMC) approach (Niederreiter, 2010) and sparse grid integration/interpolation (Gerstner & Griebel, 1998), offer faster rates of convergence but suffer from the curse of dimensionality – the number of samples \( m \) required for accurate approximations of the statistics scales exponentially with the dimensionality of \( \theta \).

A common approach is to use the \( m \) samples as a training set \( T = \{(\theta_i, u_i)\} \) for \( i = 1, 2, \ldots, m \) to instead construct an approximate mapping from \( \theta \) to \( u \). As with our previous discussion of surrogate models in Chapter 1, the approximation is considerably cheaper to evaluate than the simulation itself. As a result, the approximation may instead be used to obtain approximations for the statistics of the QOI. One of the most widely adopted methods for constructing approximations in UQ is the use of polynomial chaos (PC) expansions (Spanos & Ghanem, 1989; Xiu & Karniadakis, 2002), popular in large part thanks to the ease with which statistical moments of \( u \) may be extracted from the approximation.

A PC expansion for \( u \) as a function of the random variables \( \theta \) has the form:

\[
 u(\theta) \approx u^P(\theta) = \sum_{i=0}^{P-1} \alpha_i \psi_i(\theta), \tag{6.1}
\]
where $\alpha_i$ are unknown deterministic coefficients and in practice we truncate the expansions at some order $p$ such that the total number of basis functions is:

$$P = \frac{(p + n)!}{p!n!}. \quad (6.2)$$

The PC basis functions are constructed from tensor products of univariate polynomials, are orthonormal, and have the following properties:

1. $\psi_0(\theta) = 1$
2. $\langle \psi_i(\theta) \rangle = 0$ for $i > 0$
3. $\langle \psi_i(\theta) \psi_j(\theta) \rangle = \delta_{i,j}$ where $\delta_{i,j}$ is the Kronecker delta function

When the inputs $\theta$ follow a uniform or Gaussian distribution, PC expansions use Legendre or Hermite polynomials, respectively.

There are a number of ways to calculate the unknown coefficients in the PC expansion above, including stochastic projection, regression, and compressive sampling. Stochastic projection computes the values of the PC coefficients by using the orthonormality property of the polynomials and solving a multidimensional integral using various quadrature schemes or even MC sampling (Millman et al., 2004). Using regression, we seek an expansion that satisfies the following relationship for our $m$ samples:

$$u_j = u^P(\theta_j) = \sum_{i=0}^{P-1} \alpha_i \psi_i(\theta_j) \quad \text{for} \quad j = 1, \ldots, m. \quad (6.3)$$

In matrix form, the undetermined coefficients can be found by solving:

$$\Psi \alpha = u, \quad (6.4)$$

where $\Psi \in \mathbb{R}^{m \times P}$ contains evaluations of the $P$ normalized PC basis functions at the $m$ samples, $\alpha \in \mathbb{R}^P$ is a vector of undetermined PC coefficients, and $u \in \mathbb{R}^m$ is a vector of the $m$ corresponding QOIs. Although traditional least-squares regression techniques will accurately compute the coefficients for $m > P$ (Blatman & Sudret, 2011), when this condition is not true the system is undetermined and has an infinite number of solutions. A minimum norm least squares procedure typically results in overfitting.

When the computational cost of evaluating the model output $u(\theta)$ results in a limited set of available samples for the model output, the theory for compressive sampling indicates that it is still possible to accurately reconstruct the response from incomplete
random observations by sampling at a lower rate than required by the Shannon-Nyquist
sampling theorem (Candes et al., 2006). In the context of PC expansions, this permits
the unknown coefficients to be determined when the number of samples is less than the
cardinality of the output, i.e. $m \ll P$. In particular, a unique solution to the underde-
termined system of equations in (6.3) may be found by imposing a form of regularization
while attempting to solve the following problem:

\[
\text{Minimize: } \|\alpha\|_0 \quad \text{Subject to: } \|\Psi \alpha - u\|_2 \leq \delta.
\]

Similar to our previous discussion of dictionary learning in Chapter 2, we may seek
to solve this problem and obtain a sparse PC approximation through convex relaxation
via the $\ell_1$ norm. Peng et al. (2014) recently demonstrated a weighted $\ell_1$ approach for
PC expansions and compared it to classical $\ell_1$ minimization. Alternatively, we may solve
(6.5) by using a greedy algorithm like Orthogonal Matching Pursuit (OMP). With a basis
function dictionary composed of the $P$ polynomials, our GGDL algorithm from Chapter
5 may be adapted to build PC approximations. To apply GGDL-CV to a PC expansion
problem, we follow the procedure in Algorithm 5 by first computing the regressor matrix
$A = \Psi$ in full and then using splits of the data to obtain an optimal training error $\delta_{opt}$
at which to stop the thread which uses the whole dataset. To apply GGDL-R, we follow
the procedure in Algorithm 6 and instead compute a random subset of the vectors in the
regressor matrix $A = \Psi$ on the fly at each iteration.

We will demonstrate an application of GGDL-CV to UQ by constructing a PC expa-
sion for the thermal driven cavity flow problem with a stochastic temperature boundary
condition (LeMaitre et al., 2002; LeMaitre & Knio, 2010; Quéré, 1991). Figure 6.18
shows the geometry and boundary conditions from this problem – $T_h$ is the fixed hot
boundary, $T_c$ is the stochastic cold boundary. For a detailed overview of the problem, we
refer the reader to Peng et al. (2014). Similar to Peng et al. (2014), we seek to accurately
approximate the first two statistical moments of the vertical velocity component of the
flow field at the spatial coordinates $(0.25,0.25)$.

To obtain the full dataset, we solve the flow problem for a collection of 1000 Sobol-
sampled instances of $\theta$ on the interval $\theta_i \in [-1,1]$ for all $i = 1, \ldots n$ where in this case
$n = 20$. To obtain the 1000 quantities of interest $u(\theta)$, the flow problem is solved using
quartic Hermite elements and segmented solutions (Holdeman & Kim, 2010). The full
1000-point dataset has the following mean and variance for the quantity of interest:
\[
\mu_{(m=1000)} = -5.706E - 4 \quad \sigma^2_{(m=1000)} = 3.544E - 6 \quad .
\] (6.7)

UQ techniques attempt to recover accurate approximations for the quantity of interest with only a small number of evaluations of the expensive experiment/simulation. Therefore our goal now is to construct approximations for the statistical moments of \( u(\theta) \) using only a small subset of the full 1000-point dataset and compare them to the mean and variance of \( u \) of the full dataset.

We may treat a subset of \( m \) points from the full dataset as the training set and construct a full PC expansions with polynomial order \( p \) using the regression technique discussed earlier. To avoid overfitting, however, we are constrained to a polynomial order \( p \) for which the total number of basis functions \( P \) (see (6.2)) does not exceed the size of the training set, \( m \). With the same training set, we may construct sparse PC expansions using our GGDL algorithm but without the aforementioned constraint on the polynomial order \( p \). In theory, the use of higher order polynomials selected by the greedy procedure should yield more accurate estimates of the statistical moments of \( u \).

As an illustrative example, consider the first \( m = 100 \) points from the full dataset as a training set. A Quasi Monte Carlo approximation on this training set, for example, has the following mean and variance for the quantity of interest:
\[
\mu_{(QMC,m=100)} = -5.979E - 4 \quad \sigma^2_{(QMC,m=100)} = 3.384E - 6 \quad .
\] (6.8)
A full polynomial chaos (FPC) expansion for \( m = 100 \) points can be of either polynomial order \( p = 0 \) (\( P = 1 \)), or polynomial order \( p = 1 \) (\( P = 21 \)). The FPC expansion with \( p = 1 \) has the following expression: 
\[
\mu^{P=21}(\theta) = \alpha_0 + \alpha_1\theta_1 + \alpha_2\theta_2 + \cdots + \alpha_{20}\theta_{20}.
\]
Solving [6.4] for \( \alpha \) yields the following mean and variance for the quantity of interest:
\[
\mu_{FPC,m=100,p=1} = -5.795E - 4 \quad \sigma^2_{FPC,m=100,p=1} = 3.395E - 6 .
\] (6.9)

We may now use our GGDL-CV algorithm to construct sparse polynomial chaos (SPC) expansion for this \( m = 100 \) point dataset. To demonstrate the algorithm’s capabilities, we choose a high polynomial order – in this case \( p = 5 \). The resulting regressor matrix has a total of \( P = 53,130 \) columns representing the 53,130 basis function candidates. Using GGDL-CV with 4-fold cross validation, the algorithm halts after \( t = 14 \) iterations. Table [6.7] lists the basis functions chosen at each iteration of the algorithm, with the accompanying training set residual. It should be noted that polynomials of second and third order are used to approximate \( u \), something that an FPC expansion could not handle for this problem without overfitting.

Table 6.7: A dataset consisting of \( m = 100 \) Sobol-sampled points for the thermal driven cavity flow problem is approximated using the 4-fold GGDL-CV algorithm to construct a sparse polynomial chaos expansion. The algorithm halts at \( t = 14 \) iterations after detecting a training set residual lower than \( \delta_{opt} = 0.0457 \). The basis functions chosen at each iteration, as well as the training set residual following the iteration, are listed here.

<table>
<thead>
<tr>
<th>iteration</th>
<th>basis function</th>
<th>training set residual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \theta_2 )</td>
<td>0.7172</td>
</tr>
<tr>
<td>2</td>
<td>( \theta_3 )</td>
<td>0.5076</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0.4148</td>
</tr>
<tr>
<td>4</td>
<td>( \theta_4 )</td>
<td>0.3284</td>
</tr>
<tr>
<td>5</td>
<td>( \theta_1 )</td>
<td>0.2214</td>
</tr>
<tr>
<td>6</td>
<td>( \theta_5 )</td>
<td>0.1690</td>
</tr>
<tr>
<td>7</td>
<td>((3\theta_2^2 - 1)/2)</td>
<td>0.1452</td>
</tr>
<tr>
<td>8</td>
<td>( \theta_2\theta_3 )</td>
<td>0.1178</td>
</tr>
<tr>
<td>9</td>
<td>( \theta_6 )</td>
<td>0.0901</td>
</tr>
<tr>
<td>10</td>
<td>( \theta_7 )</td>
<td>0.0754</td>
</tr>
<tr>
<td>11</td>
<td>( \theta_2\theta_4 )</td>
<td>0.0648</td>
</tr>
<tr>
<td>12</td>
<td>( \theta_8 )</td>
<td>0.0546</td>
</tr>
<tr>
<td>13</td>
<td>( \theta_2\theta_6 )</td>
<td>0.0473</td>
</tr>
<tr>
<td>14</td>
<td>( \theta_2\theta_5\theta_1 )</td>
<td>0.0421</td>
</tr>
</tbody>
</table>

This sparse PC expansion has the following mean and variance for the QOI:
\[
\mu_{SPC,m=100,p=5} = -5.735E - 4 \quad \sigma^2_{SPC,m=100,p=5} = 3.532E - 6 .
\] (6.10)
Compared to the results of QMC in (6.8) and FPC in (6.9), these results are much more accurate approximations of the $m = 1000$ dataset mean and variance in (6.7). In addition, the SPC expansion uses 7 fewer basis functions than the FPC expansion of order $p = 1$. Table 6.8 provides a collection of results for QMC, FPC, and SPC over various sample sizes $m$ and polynomial orders $p$. Errors are calculated as a relative error from the mean and variance of a 3000-point dataset. Apart from a couple of cases of overfitting for the higher order expansions, GGDL-CV is able to construct sparse expansions which are more accurate than QMC and FPC.

Table 6.8: Uncertainty quantification for the thermal driven cavity flow problem is performed using three different techniques - Quasi Monte Carlo (QMC), full polynomial chaos expansions (FPC), and sparse polynomial chaos expansions (SPC). The mean and variance of the vertical velocity component at $(0.25,0.25)$ are computed using smaller datasets of 50, 100, and 250 points. To compute error, they are compared to the mean and variance from a larger dataset of 3000 points. Both FPC and SPC expansions are built for a variety of polynomial orders, $p$, each resulting in a different number of total basis functions, $P$. FPC expansions are built using a least squares solution. SPC expansions are built using 10-fold GGDL-CV and their results are averaged over 5 runs. Bold results denote SPC outperforming FPC.

<table>
<thead>
<tr>
<th>$m$</th>
<th>QMC $\mu$ error</th>
<th>FPC $\mu$ error</th>
<th>SPC $\mu$ error</th>
<th>QMC $\sigma^2$ error</th>
<th>FPC $\sigma^2$ error</th>
<th>SPC $\sigma^2$ error</th>
<th>sparsity</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.14921</td>
<td>0.02320</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.02498</td>
<td>0.09287</td>
<td>0.03208</td>
<td>0.04624</td>
<td>0.03133</td>
<td>0.02823</td>
<td>11.2</td>
</tr>
<tr>
<td>2</td>
<td>0.77202</td>
<td>0.63930</td>
<td>0.03133</td>
<td>0.04624</td>
<td>0.03133</td>
<td>0.02823</td>
<td>11.2</td>
</tr>
<tr>
<td>3</td>
<td>0.91705</td>
<td>0.95741</td>
<td>0.27460</td>
<td>0.03280</td>
<td>0.02746</td>
<td>0.03280</td>
<td>10.0</td>
</tr>
<tr>
<td>4</td>
<td>0.98748</td>
<td>0.99092</td>
<td>0.05494</td>
<td>0.04149</td>
<td>0.05494</td>
<td>0.04149</td>
<td>8.2</td>
</tr>
<tr>
<td>5</td>
<td>0.99357</td>
<td>0.99776</td>
<td>0.08626</td>
<td>0.01390</td>
<td>0.08626</td>
<td>0.01390</td>
<td>16.6</td>
</tr>
</tbody>
</table>

| 100  | 0.03778          | 0.04580         |                 |                      |                      |                      |         |
| 1    | 0.00590          | 0.04249         | 0.00352         | 0.04928              | 0.00352              | 0.04928              | 8.6     |
| 2    | 0.40909          | 0.42417         | 0.00473         | 0.00187              | 0.00473              | 0.00187              | 23.0    |
| 3    | 0.87956          | 0.91856         | 0.00432         | 0.00174              | 0.00432              | 0.00174              | 17.0    |
| 4    | 0.97764          | 0.98367         | 0.00631         | 0.00366              | 0.00631              | 0.00366              | 15.8    |
| 5    | 0.99126          | 0.99611         | 0.00549         | 0.00503              | 0.00549              | 0.00503              | 16.8    |

| 250  | 0.02260          | 0.00707         |                 |                      |                      |                      |         |
| 1    | 0.00416          | 0.00913         | 0.00420         | 0.00902              | 0.00420              | 0.00902              | 8.0     |
| 2    | 0.00102          | 0.01587         | 0.00065         | 0.00197              | 0.00065              | 0.00197              | 43.2    |
| 3    | 0.77598          | 0.79288         | 0.00063         | 0.00032              | 0.00063              | 0.00032              | 43.8    |
| 4    | 0.95545          | 0.96135         | 0.00105         | 0.00076              | 0.00105              | 0.00076              | 31.4    |
| 5    | 0.98666          | 0.99091         | 0.00071         | 0.00329              | 0.00071              | 0.00329              | 24.0    |

6.3.2 The “Gappy” Data Problem

Our second case study tackles a practical problem with applications in a number of fields. We attempt to fill voids in a set of “gappy” data from a multi-dimensional input-output relationship. The problem we tackle in this case study comes from the field of gas turbine
combustor design, where it is important to have accurate measurements of the velocity field at various operating conditions (Huang & Yang, 2009). Particle image velocimetry (PIV) is one of the most common techniques for measuring the velocity field of gas turbine combustors. However, gaps in the processed velocity fields may arise from combining low pulse energy lasers with high pressure flames or from non-uniform particle seeding. To make use of the gappy dataset, it is important to accurately approximated the missing values.

A PIV dataset consisting of three velocity components was obtained from Saini et al. (2016). Data for each velocity component consists of 7870 time steps where each time step includes values on a two-dimensional spatial grid \((x, y)\) of 56\(\times\)44 points. We refer the reader to the Saini et al. (2016) paper and references within for a detailed description of the combustor and PIV system. Figure 6.19 illustrates the full PIV data for the first time step. The individual magnitudes of the three velocity components are plotted separately to better illustrate that the full dataset consists of 7870 time steps, each containing three velocity values for the full spatial grid.

In addition to the PIV dataset, the authors provided us with their artificial gaps in the form of masks for the dataset. These gaps vary in size and location and are added to the dataset in a manner similar to the gaps found in real experimental data. Figure 6.20 illustrates an example of both the full and gappy data for two of the three velocity components at three time steps: 1, 5, and 10. The third velocity component is excluded for clarity. The three figures of the full, non-gappy, flow (a, c, e) clearly demonstrate an evolution in the velocity components with time. The gappy figures (b, d, f) show the same data but with changing, block-like, gaps at locations for which the data is masked.

Our goal is to compare our GGDL algorithm to two popular methods for gap-filling: Kriging (Gunes et al. 2006) and gappy proper orthogonal decomposition (GPOD) (Evenson & Sirovich 1995; Raben et al. 2012). We will train GGDL predictors using the non-masked data points and compute a test error based on the masked data points. Given a vector \(v\) of \(m_v\) masked values and the predicted values \(\hat{v}\), we will compute the test error as follows:

\[
\text{test error} = \frac{\|v - \hat{v}\|_2}{\|v\|_2}.
\] (6.11)

We will treat each individual velocity component as an independent output. Before comparing GGDL to the two other gap-filling methods, some thought was given to how the approximation problem should be approached. The size and dimensionality of the problem ultimately depends on how we choose to structure the training dataset. We gave consideration to three approaches:
Figure 6.19: Three velocity components of PIV data for the first time step are plotted individually for their spatial coordinates. The full dataset consists the three velocity components on a spatial grid for 7870 time steps.

1. As a collection of 7870 individual 2-dimensional approximation problems for each velocity component. For each problem, the velocity is modelled as a function of the two spatial coordinates.

\[ v = f(x, y) \]

For each problem, the dataset consists of at most \( m = 56 \times 44 = 2464 \) points of dimensionality \( n = 2 \). Because the size of the gap determines the size of the training and test sets, only an upper bound on \( m \) can be estimated.

2. As a single 3-dimensional approximation for each velocity component. For each problem, the velocity is modelled as a function of the two spatial coordinates and...
Figure 6.20: Full and gappy data for the first and second velocity components is shown in a quiver plot for three time steps: 1, 5, and 10. For clarity, component 3 is excluded.
time. Theoretically, this may perform better than the first approach because time also included as a variable.

\[ v = f(x, y, t) \]

For each problem, the dataset consists of at most \( m = 56 \times 44 \times 7870 = 19,391,680 \) points of dimensionality \( n = 3 \).

3. As a collection of smaller 3-dimensional problems for each velocity component, where data from only the nearest \( T \) time steps is used in the training set. This “moving time window” of \( T \ll 7870 \) can be used to reduce the size of the previous approach while potentially maintaining the benefits of including the time variable.

\[ v = f(x, y, t) \]

For each problem, the dataset consists of at most \( m = 56 \times 44 \times T \) points of dimensionality \( n = 3 \).

We compared the three approaches by using the first 500 time steps of the first velocity component and approximating the dataset as 500 2-dimensional problems (approach 1), a single 3-dimensional problem (approach 2), and 50 3-dimensional problems (approach 3, \( T = 10 \)). We found that, using our GGDL-R algorithm with a sub-dictionary size of 60 basis functions, the first approach provided the most accurate predicted values at the gappy data sites. Henceforth, it is assumed that any reported GGDL-R results are based on the 2-dimensional variant of the problem.

We compared test errors between Kriging, GPOD, and with our GGDL-R algorithm. Results for Kriging and GPOD were obtained from (Saini et al., 2016). The GGDL-R algorithm used the following options:

- A basis function dictionary consisting of 7 Gaussian RBFs of various shapes centred at the training examples
- A random sub-dictionary of size 60
- A maximum limit of 350 iterations
- A lower bound on the training residual of 0.04

Figure 6.21 shows quiver plots for the original dataset and the three approximations. A single time step is chosen and the gappy data is presented in a different colour. Once again it is the first two velocity components which are shown. Visual inspection suggests
that all three methods perform well and approximate the data gaps with considerable accuracy. Numerically, however, we found that GPOD is the most accurate of the three methods, followed by the Kriging approximation, and then our GGDL-R algorithm. For all 7870 time steps of the second velocity component, for example, GPOD had an average test error of 0.1268, Kriging had an average test error of 0.1375, and GGDL-R averaged 0.1492.

Figure 6.21: Three approximation methods (GPOD, Kriging, and GGDL-R) are used to approximate the velocity field at the first time step. The missing data from Figure 6.20 is shown in red in the original field (a). Predictions for the missing data are also shown in red for the three methods (b-d). For clarity, these quiver plots exclude the third velocity component and are zoomed in to a $30 \times 25$ grid.

Although, averaged over all time steps, GGDL-R was perhaps the least accurate of the three approximation techniques for this problem, we must mention that GGDL-R is significantly faster that Kriging at constructing and evaluating approximations for the
gappy data – 62 hours over the entire dataset for GGDL-R versus 86 hours for Kriging. GPOD results were collected by a different group without consideration for training times. At the same time, GGDL-R actually provided the best predictor for 2519 of the 7870 time steps. Figure 6.22 shows a probability density function of the errors in approximations of the second velocity component for all time steps. The GGDL-R distribution appears to peak slightly left of the Kriging distribution, but suffers from a higher average error due to a “tail” towards the higher error values. With its higher variance of errors, GGDL-R actually records more instances of error below 10% than Kriging and more instances of error below 7% than both GPOD and Kriging. At the same time, it exhibits more instances of error above 17.5% than both methods as well. Finally, we must note that the predictors constructed by GGDL-R for this dataset had an average sparsity of 284 basis functions – about one fifth of the Kriging average sparsity of 1547 basis function.

![Figure 6.22: Probability density function (PDF) of test errors in the approximation of the second velocity component by GPOD, Kriging, and GGDL-R. The average test errors were 0.1268, 0.1375, and 0.1492, respectively.](image)

6.4 Summary and Concluding Remarks

We began this chapter with a thorough demonstration of the capabilities of our GGDL algorithm. First, the evolution of a predictor was illustrated for both the parallel cross validation-based algorithm (GGDL-CV) and the randomized algorithm (GGDL-R). A collection of examples, mostly from synthetic data, was then used to illustrate three interesting capabilities of GGDL: (i) an ability to “tune” RBF shape parameters by discretizing the possible shape parameter values into a collection of RBF candidates,
(ii) an ability to use non-training data centred RBFs, and (iii) an ability to construct predictors with a variety of kernel types.

Following this collection of demonstrative examples, we compared our GGDL algorithm to Gaussian process models (GPM) and support vector machines (SVMs) – two popular machine learning techniques for regression. We used four popular machine learning datasets (Abalone, Boston, CPU Small, and Diabetes) to show that GGDL constructs predictors which are as accurate as, and sometimes more accurate than, the two competing state-of-the-art algorithms. GGDL-CV continuously produced the most sparse predictors while GGDL-R was also able to produce predictors which were considerably more sparse than GPM or SVM.

Finally, we used our GGDL algorithm to tackle two case studies. First, we showed how GGDL-CV may be used to construct sparse polynomial chaos (PC) expansions for uncertainty quantification. We constructed a sparse PC expansion for the thermal driven cavity flow problem with a stochastic temperature boundary condition and showed that GGDL-CV was able to construct incredibly sparse PC expansions with exceptional accuracy. Our second case study tackled the problem of filling voids in “gappy” data. We compared GGDL-R to two popular methods for data gap filling – Kriging and gappy proper orthogonal decomposition (GPOD). The specific practical problem at hand was from a dataset of particle image velocimetry (PIV) measurements used in gas turbine combustor design. We discussed a number of ways that GGDL could be used to approximate the dataset for gap filling and finally showed that the GGDL-R approximations are comparable to Kriging in terms of accuracy, but are significantly cheaper to construct and evaluate.
Chapter 7

Greedy Algorithms for Gradient-Enhanced Data

In this chapter, we concern ourselves with approximation problems for which the dataset contains not only the typical input-output relationship, but also some information on the sensitivity of the output – the “gradients”. Our goal is to demonstrate that our greedy algorithms can be quickly adapted to deal with this new type of data. We present numerical studies on a set of problems to show that using gradient data can enhance the quality of predictors built using a greedy procedure.

We begin by discussing our motivation for tackling these types of problems. After that, we outline our method for incorporating sensitivity data into our algorithms. We present two different greedy “Hermite” algorithms which are capable of using gradient information during approximation. These two methods are based largely on the greedy framework developed in Chapter 3. Finally, we present a section in which we compare the performance between the two gradient-enhanced algorithms and compare them to our GRBF algorithm from Chapter 3.

7.1 Motivation

We are motivated by approximation problems for which some information on the sensitivity of the output is available i.e., the partial derivatives of some output objective $y$ with respect to an $n$-dimensional vector of independent inputs $x$. Typically it is derivatives of the first order which that are the cheapest to compute, which we will denote as $\nabla y$:

$$\nabla y = \left[ \frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \ldots, \frac{\partial y}{\partial x_n} \right]^T \in \mathbb{R}^n. \quad (7.1)$$
One of the most useful features of our surrogate modelling algorithms is that they can be applied to problems from any field since the source of data does not matter. This holds true even when sensitivity data is added to the problem. Nonetheless, we will provide a brief summary of a number of ways in which sensitivities may be calculated in practical situations, as these techniques may be used to “enhance” a typical dataset with sensitivity data. Keane & Nair (2005) devote a chapter to discussing methods for computing sensitivities in both algebraic and functional analysis settings.

Adjoint methods and direct methods are two ways to calculate sensitivities. Both methods are efficient and capable of computing exact sensitivities, but require an understanding of the problem’s governing equations. For more on direct methods, we refer the reader to Haftka & Gurdal (1992) for a direct sensitivity procedure applied to nonlinear structural systems. For problems with a large number of dimensions $n$, direct methods prove to be inefficient and adjoint methods are instead employed. In practice, adjoint methods have been in use for quite some time in nuclear engineering (Cacuci, 1981), control theory (Lions, 1971), and computational fluid dynamics (Jameson, 1988), Rumpfkeil & Zingg, 2010) (Truong et al., 2008).

We are interested in the class of problems for which it is feasible to efficiently compute the partial derivatives – examples include problems governed by PDEs where the adjoint method may be used. Hermite interpolation is the idea which we will use to incorporate gradient data into surrogate models. The theoretical foundations for Hermite interpolation in the context of radial basis function approximation can be found in the literature (Zhongmin, 1992; Fasshauer, 1999; Narcowich & Ward, 1994; Ong et al., 2008). Gradient enhanced Kriging is perhaps the best-established method, but suffers from high computational cost and memory requirements (Liu, 2003; Han et al., 2013; Chung & Alonso, 2002). Let us begin by assuming that we have a training dataset of $m$ $n$-dimensional examples, the corresponding $m$ function values, and all $n$ derivatives at all $m$ examples:

$$\{ x_i, y_i, \nabla y_i \} \quad \text{where } i = 1, 2, \ldots, m,$$  \hspace{1cm} (7.2)

where $\nabla y$ is defined as in (7.1). A Hermite radial basis function interpolant has the form:

$$f(x) = \sum_{i=1}^{m} w_i \phi(x, c_i, \sigma) + \sum_{i=1}^{m} \sum_{j=1}^{n} \beta_{ij} \frac{\partial \phi(x, c_i, \sigma)}{\partial x_j},$$  \hspace{1cm} (7.3)

where $w_i$ and $\beta_{ij}$, $i = 1, 2, \ldots, m$, $j = 1, 2, \ldots, n$ are the undetermined weights. This predictor structure is similar to the classical RBF approximations we discussed in Chapter 3. In (7.3), however, we include the derivatives of the RBFs. Both the RBFs and the
derivatives are centred at the training points.

Because the dataset has \( y \) and \( \nabla y \) at \( m \) examples \( x \), we need a total of \( m(n+1) \) linear algebraic equations. Whereas in classical RBF interpolation we used \( m \) equations to construct the Gram matrix \( K \in \mathbb{R}^{m \times m} \), in Hermite RBF interpolation we now have a much larger Gram matrix \( K \in \mathbb{R}^{m(n+1) \times m(n+1)} \) formed by \( m \) equations for the function values and an additional \( mn \) equations for the derivative information. The Gram matrix can be structured in the following block form:

\[
K = \begin{bmatrix}
\Phi_{11} & \Phi_{12} & \ldots & \Phi_{1m} \\
\Phi_{21} & \Phi_{22} & \ldots & \Phi_{2m} \\
\vdots & \vdots & \ddots & \vdots \\
\Phi_{m1} & \Phi_{m2} & \ldots & \Phi_{mm}
\end{bmatrix},
\tag{7.4}
\]

where each block is organized as follows:

\[
\Phi_{ij} = \begin{bmatrix}
\phi(x_i, c_j, \sigma) & \frac{\partial \phi(x_i, c_j, \sigma)}{\partial x_1} & \ldots & \frac{\partial \phi(x_i, c_j, \sigma)}{\partial x_{n}} \\
\frac{\partial \phi(x_i, c_j, \sigma)}{\partial x_1} & \frac{\partial^2 \phi(x_i, c_j, \sigma)}{\partial x_1^2} & \ldots & \frac{\partial^2 \phi(x_i, c_j, \sigma)}{\partial x_1 \partial x_n} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial \phi(x_i, c_j, \sigma)}{\partial x_n} & \frac{\partial^2 \phi(x_i, c_j, \sigma)}{\partial x_n \partial x_1} & \ldots & \frac{\partial^2 \phi(x_i, c_j, \sigma)}{\partial x_n^2}
\end{bmatrix}.
\tag{7.5}
\]

Immediately notable is the high computational cost and memory requirements of Hermite RBF interpolation for large \( m \) or \( n \). At the same time, it is arguably even more difficult to tune the global shape parameter \( \sigma \) since it now affects not only the \( m \) basis functions \( \phi \) but also their \( mn \) derivatives \( \partial \phi / \partial x \). The high computational cost and memory requirements associated with Hermite RBF interpolation may be reduced by selectively using gradient data at only a subset of the examples. This however, assumes that there exists some prior knowledge of the problem, which is not always the case. At the same time, it makes the engineer/developer/scientist responsible for screening the data. Ideally we would like an algorithm to be given all of the data and, if sparsity is desirable, it would select the necessary basis functions to construct a predictor that generalizes well. Similarly \( \sigma \) may be tuned using cross-validation or heuristic methods, neither of which is ideal, as we have already shown in Chapter 3.

Motivated by a desire to approximate gradient-enhanced datasets, we saw that Hermite RBF approximations suffer from the same problems as classical RBF approximations in the non-derivative setting – namely difficulties in tuning the global shape parameter \( \sigma \) and high cost. In this chapter we will look to adapt our greedy surrogate modelling algorithms to tackle approximation problems with gradient data. In the next section we
begin by explaining how to incorporate gradient data into our approximation framework.

### 7.2 Incorporating Gradient Data

To understand how we would incorporate gradient data in our approximations, we begin by revisiting the form of the training dataset used in our regular approximation problems. When gradient data was not available:

$$T_{\text{reg}} := \{(x_i, y_i)\} \quad \text{where } i = 1, 2, \ldots, m.$$  \hfill (7.6)

When gradient data is available, we consider the most general of cases. Unlike (7.2), we impose no requirement for gradient data to be specified at all training data examples \(\{x_i\}_{i=1}^m\) or to be specified for all training data dimensions \(\{x_j\}_{j=1}^n\). We will therefore abandon the \(\nabla y\) notation from the previous section and instead let \(y_i\) denote any type of output (function value or gradient) corresponding to an example \(x_i\). To differentiate between function values and gradients, we instead introduce a third quantity. Let us assume we have a collection of \(g\) gradient data records in the following form:

$$T_{\text{grad}} := \{(x_i, y_i, \Delta_{\{i\}})\} \quad \text{where } i = 1, 2, \ldots, g,$$  \hfill (7.7)

where \(\Delta_{\{i\}} \in \mathbb{N}\) specifies the dimension of \(x\) with respect to which \(y_i\) is a gradient, i.e. if \(\Delta_{\{i\}} = j\) then,

$$y_i = \frac{\partial y}{\partial x_j} \bigg|_{x = x_i}.$$  \hfill (7.8)

For example, if \(\Delta_{\{4\}} = 2\), then \(y_4\) corresponds to \(\partial y/\partial x_2\) at \(x_4\). We refer to this vector of gradient directions \(\Delta = [\Delta_{\{1\}}, \Delta_{\{2\}}, \ldots, \Delta_{\{g\}}]^T\) as the “gradient keys”.

An approximation problem with gradient data will have a training dataset with examples of both forms: \(T_{\text{reg}}\) and \(T_{\text{grad}}\). It is thus necessary to also include gradient keys for non-derivative data. The logical choice is \(\Delta_{\{i\}} = 0\) for all \(m\) examples in \(T_{\text{reg}}\). We may now express a complete gradient-enhanced training dataset as:

$$T := \{(x_i, y_i, \Delta_{\{i\}})\} \quad \text{where } i = 1, 2, \ldots, m + g,$$  \hfill (7.9)

where \(x_i \in \mathbb{R}^n, y_i \in \mathbb{R},\) and \(\Delta_i \in \mathbb{W}\). For clarity, we will also designate the total training set size \(M = m + g\).

When computing a regressor matrix column \(\phi \in \mathbb{R}^M\) associated with some basis function \(\phi(x)\), we can use \(\Delta\) to calculate the appropriate value or derivative of \(\phi(x)\) for each element in the column. We evaluate \(\phi(x)\) for each training set example where \(\Delta = 0\).
and differentiate \( \phi(x) \) with respect to \( x_\Delta \) for all examples where \( \Delta > 0 \). If the training set is ordered such that the first \( m \) examples have \( \Delta = 0 \) and the other \( g \) examples have \( \Delta > 0 \), then computing the column for \( \phi(x) \) is rather straightforward,

\[
\phi = \begin{bmatrix}
\phi(x_1) \\
\vdots \\
\phi(x_m) \\
\frac{\partial \phi}{\partial x_\Delta(m+1)}(x_{m+1}) \\
\vdots \\
\frac{\partial \phi}{\partial x_\Delta(m+g)}(x_{m+g})
\end{bmatrix}.
\] (7.10)

Equipped with an understanding of the new training set forms we face when approximating gradient-enhanced data, as well as an ability to compute basis function vectors, we can now move on to developing greedy algorithms in this setting.

### 7.3 Gradient-Enhanced Greedy Algorithms

We explore two ways to use greedy dictionary learning algorithms to approximate gradient-enhanced datasets, largely based on the algorithm frameworks developed in Chapters 3 and 5. To differentiate between our algorithms from those in earlier chapters and their “Hermite” versions in this chapter, we will use a suffix “H” in the algorithm acronyms. The first “passive” method involves using the gradient training examples to affect only the least squares solution at each stage of the greedy algorithm. The second “active” method is actually capable of adding derivative basis functions to the predictor in order to better approximate the data.

To avoid redundancy, we will focus here only on gradient-enhanced greedy radial basis function algorithms (GRBFH). The gradient-enhancing techniques described in this chapter may be applied to the GGDL algorithm from Chapter 5, and although this process is not trivial, it is conceptually straightforward.

#### 7.3.1 GRBFH-P: Passively Using Gradient Information

Our first Hermite GRBF algorithm is “passive” in the sense that the gradient data does not directly affect the basis function dictionary or the candidate selection step of the greedy procedure. Referring back to Chapter 3, the parameterized dictionary of basis function \( D(\sigma) \) used in this algorithm is the same as the original GRBF algorithm. That is, it is composed of basis functions centred at each of the \( m \) training examples \( x_i \) for
which $\Delta_{(i)} = 0$. Once a basis function centre is selected, based on the location of the highest residual, the basis function shape parameter $\sigma$ is tuned as before. The problem statement we consider can be stated as follows:

Given a gradient-enhanced training set $T$ as in (7.9), a parametrized dictionary $D(\sigma)$, and target training error $\delta$, find the smallest subset of $t$ basis functions $\phi_i, i = 1, \ldots, t$ from $D$, constants $\alpha_i, i = 1, \ldots, t$, and shape parameters $\sigma_i, i = 1, \ldots, t$, if they exist, such that

$$
\sum_{i=1}^{m} \left( \sum_{j=1}^{t} \alpha_j \phi_j(x_i, c_j, \sigma_j) - y_i \right)^2 + \sum_{i=m+1}^{g} \left( \sum_{j=1}^{t} \alpha_j \frac{\partial \phi_j(x_i, c_j, \sigma_j)}{\partial x} - y_i \right)^2 \leq \delta .
$$

(7.11)

Alternatively the problem statement can be rewritten as a constrained $\ell_0$ norm minimization problem of the form:

$$
\text{Minimize: } ||\alpha||_0
$$

$$
\text{Subject to: } ||K(\sigma)\alpha - y||_2 \leq \delta ,
$$

(7.12)

(7.13)

where, as usual, $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}^T$ is the vector of weights, $\sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}^T$ is the vector of shape parameters, and $K(\sigma) \in \mathbb{R}^{M \times m}$ is a $\sigma$-parameterized Gram matrix of the $m$ basis functions centred at the training data. However for GRBFH-P, the output vector $y$ is of length $M$ and contains both function values and gradients.

The major difference between our previous algorithms and GRBFH-P is in the computation of basis function columns at each iteration. At iteration $t$ of GRBF, the index of the chosen RBF is designated $i^{(t)}$ and the corresponding column of the Gram matrix was expressed as $K(:, i^{(t)})$. Similarly, at iteration $t$ of GGDL, the column of the regressor matrix corresponding to the chosen basis function was expressed as $\phi_{i^{(t)}}$. In both cases, the column was computed by simply evaluating the chosen basis function at each training example $x_i$ in $T$. In GRBFH-P, however, the basis functions are now evaluated at training examples for which gradient information is available. We therefore use (7.10) to compute the columns for each basis function. To accomplish this, the basis function candidates must be at least once differentiable and have pre-defined derivative expressions which are calculated for all examples where $\Delta_{(i)} > 0$. In the case of a Gaussian RBF $\phi_i(x)$ centred at $c_i$, for example, the following first derivative expression is used:

$$
\frac{\partial}{\partial x_k} \phi_i(x) = -\frac{2}{\sigma} (x_k - c_{i,k}) \exp \left(-\frac{||x - c_i||^2}{\sigma}\right)
$$

(7.14)

It then follows that during the QR update step of the algorithm, the least squares
solution is computed in a way which reduces the residual for all $y_i$ in $T$, inclusive of the gradient information. The result is an algorithm which builds a predictor composed of the same RBF candidates as GRBF, but this time with consideration for the gradient information contained in the training dataset.

Algorithm 7 outlines the GRBFH-P approximation procedure. The algorithm is nearly identical to Algorithm 4. The only difference is the change in the starting set $I^{(0)}$, which is no longer set to $\emptyset$. In order to eliminate the gradient example centres from the candidate set, $I$ is immediately set to hold the indices of all training examples for which $\Delta_{(i)} > 0$. The rest of the algorithm proceeds as before, with the notable fact that the basis function columns are now of length $M = m + g$ and (7.10) is used to compute $K(:, i^{(t)})$.

**Algorithm 7** Greedy QR procedure for passively approximating gradient-enhanced datasets using RBF dictionaries.

**Inputs:**
- Gradient-enhanced training dataset $T$,
- Dictionary of radial basis functions $D$,
- Stopping criteria ($t_{\text{max}}$, $\delta_{\text{min}}$, etc).

**Outputs** (at iteration $t$):
- Indices of chosen basis functions $I^{(t)}$,
- Tuned shape parameters $\sigma_1, \ldots, \sigma_t$,
- Vector of weights $\alpha^{(t)}$.

1. $t \leftarrow 0$, $I^{(0)} \leftarrow \{j \forall \Delta_{(j)} > 0\}$, $r^{(0)} \leftarrow y$, $Q^{(0)} \leftarrow [\ ]$, $R^{(0)} \leftarrow [\ ]$.

repeat

2. $t \leftarrow t + 1$.
3. Choose RBF centre where residual is highest: $i^{(t)} = \arg \max_j \nabla_{(j)} \left | r_j^{(k-1)} \right |$.
4. Update set of chosen basis functions: $I^{(t)} \leftarrow I^{(t-1)} \cup i^{(t)}$.
5. Calculate $K(:, i^{(t)})$ using (7.10).
6. Tune shape parameter for $K(:, i^{(t)})$: $\sigma_{i^{(t)}} = \arg \min ||r||_2$.
7. Update $Q^{(t-1)}$ and $R^{(t-1)}$ to $Q^{(t)}$ and $R^{(t)}$ with $K(:, i^{(t)})$.
8. Update residual: $r^{(t)} \leftarrow r^{(t-1)} - \left(Q^{(t)}(:, t)\right)^T y Q^{(t)}(:, t)$.
9. Calculate stopping criteria.

until Stopping criteria are met.

10. Solve for weights from the system: $R^{(t)} \alpha^{(t)} = \left(Q^{(t)}\right)^T y$. 
7.3.2 GRBFH-A: Actively Adding Gradient Basis Functions

Our second algorithm for gradient-enhanced dataset is “active” in the sense that it is capable of adding Hermite basis functions. These basis functions are added in response to errors in the gradient records of the training set.

Unlike GRBFH-P, this algorithm, GRBFH-A, includes all training example centres as candidates. Therefore, the set of chosen indices \( \mathcal{I} \) begins as the empty set, as before in GRBF. Let us consider an example where the training set consists of \( m \) regular function value records as well as all \( n \) derivatives at the same \( m \) examples – for a total of \( g = mn \) gradient records and a training set of size \( M = m + g = (n + 1)m \). Whereas GRBFH-P would approximate this dataset using a dictionary of only \( m \) basis functions centred at the \( m \) examples where \( \Delta = 0 \), GRBFH-A expands the dictionary to the following \( M \) candidates:

\[
\mathcal{D} = \{ \phi_1(x), \ldots, \phi_m(x), \frac{\partial \phi_{m+1}(x)}{\partial x_{\Delta_{m+1}}}, \ldots, \frac{\partial \phi_{m+g}(x)}{\partial x_{\Delta_{m+g}}} \}. \tag{7.15}
\]

We once again follow the GRBF procedure of selecting candidates based on the location of the highest residual. At iteration \( t \), we first find the index of the highest residual, \( i(t) \). In the case where the residual is highest at a training example for which \( \Delta_{i(t)} = 0 \), we follow the previous procedure of using (7.10) to compute the basis function column \( K(:,i(t)) \). However, in the case where the residual is highest for a gradient example (\( \Delta_{i(t)} > 0 \)), a Hermite basis function is instead added to the predictor. In this case, the Hermite basis function is the derivative of \( \phi \) with respect to \( x_{\Delta_{i(t)}} \).

It should be clear at this point that when a Hermite basis function is added to the predictor, computing the column \( K(:,i(t)) \) will require calculating the second derivative of \( \phi \). This is similar to (7.10), but this time the basis function being added is itself a derivative, so:

\[
K(:, i(t)) = \begin{bmatrix}
\frac{\partial \phi}{\partial x_{\Delta_{i(t)}}}(x_1) \\
\vdots \\
\frac{\partial \phi}{\partial x_{\Delta_{i(t)}}}(x_m) \\
\frac{\partial^2 \phi}{\partial x_{\Delta_{i(t)}} \partial x_{\Delta_{m+1}}}(x_{m+1}) \\
\vdots \\
\frac{\partial^2 \phi}{\partial x_{\Delta_{i(t)}} \partial x_{\Delta_{m+g}}}(x_{m+g})
\end{bmatrix}. \tag{7.16}
\]

As an example, consider the case of using Gaussian RBFs. If an index corresponding to \( \Delta > 0 \) is selected, we may use (7.16) with the following two expressions for the second
derivative of the Gaussian RBF:

\[
\frac{\partial^2}{\partial x_k \partial x_l} \phi_i(x) = 4 \frac{\sigma^2}{\sigma^2} (x_k - c_{i,k}) (x_l - c_{i,l}) \exp \left( -\frac{\|x - c_i\|^2}{\sigma} \right),
\]

(7.17)

\[
\frac{\partial^2}{\partial x_k^2} \phi_i(x) = 4 \frac{\sigma^2}{\sigma^2} (x_k - c_{i,k}) (x_k - c_{i,k}) \exp \left( -\frac{\|x - c_i\|^2}{\sigma} \right) - 2 \frac{\sigma}{\sigma} \exp \left( -\frac{\|x - c_i\|^2}{\sigma} \right),
\]

(7.18)

where for clarity we made use of \(k\) and \(l\) instead of the \(\Delta\)-based notation above. When a Hermite Gaussian RBF is placed at \(x_{i(t)}\), we would compute its value for some other example \(x_j\) using (7.17) if \(\Delta_{\{i(t)\}} \neq \Delta_{\{j\}}\), or using (7.18) if \(\Delta_{\{i(t)\}} = \Delta_{\{j\}}\).

Algorithm 8 outlines the GRBFH-A approximation procedure. Once again, the algorithm is very similar to Algorithm 4 and Algorithm 7. The major differences here are that the set of chosen indices \(\mathcal{I}\) is once again initiated as \(\emptyset\) and we make use of both (7.10) and (7.16) to compute the columns associated with the basis functions.

### 7.3.3 Gradient-Enhanced GGDL and \(\Delta\)-Based Scaling

Modifying the GGDL-R and GGDL-CV algorithms to approximate gradient-enhanced datasets follows a similar procedure. The major difference is in the definition of the basis function dictionary \(\mathcal{D}\).

An algorithm for approximating gradient data with GGDL (GGDLH) would have a general dictionary of basis functions, not restricted to RBFs or even basis functions of the same kernel type. However, it is necessary to have a way to calculate the \(n\) derivatives of any basis function \(\phi \in \mathcal{D}\) in order to use (7.10) to compute the basis functions columns \(\phi \in \mathbb{R}^M\). For some basis functions, like the popular Gaussian, analytical expressions for \(\partial \phi / \partial x\) are available. If an expression is not available, it may be possible to use finite difference methods to compute the gradient of \(\phi\) at some location \(x\).

Because GGDL is, by definition, a general dictionary algorithm, there is no real difference between the “passive” and “active” versions of this algorithm’s gradient extension. In fact, Hermite basis functions can easily be used in our first implementation of GGDL, even though they would not be used in response to any gradient data in the training set. For this reason, we forego explicitly outlining the GGDLH algorithm and deem it sufficient to say that the only necessary change is to use the gradient keys \(\Delta_i\) to correctly compute the basis function columns.

There are two notable hurdles which must be kept in consideration when using these algorithms to approximate gradient-enhanced datasets. Firstly, caution must be exercised
Algorithm 8 Greedy QR procedure for actively approximating gradient-enhanced datasets using dictionaries with Hermite basis functions.

Inputs:
Gradient-enhanced training dataset $\mathcal{T}$,
Dictionary of radial basis functions $\mathcal{D}$,
Stopping criteria ($t_{\text{max}}$, $\delta_{\text{min}}$, etc).

Outputs (at iteration $t$):
Indices of chosen basis functions $I(t)$,
Tuned shape parameters $\sigma_1, \ldots, \sigma_t$
Vector of weights $\alpha(t)$.

1. $t \leftarrow 0$, $\mathcal{T}(0) \leftarrow \emptyset$, $r(0) \leftarrow y$, $Q(0) \leftarrow \left[ \right]$, $r(0) \leftarrow \left[ \right]$.

repeat
2. $t \leftarrow t + 1$.
3. Choose RBF centre where residual is highest: $i(t) = \arg \max_{j \notin I(t)} |r(j)|$.
4. Update set of chosen basis functions: $\mathcal{T}(t) \leftarrow \mathcal{T}(t-1) \cup i(t)$.
5. Calculate $K(:, i(t))$ using (7.10) if $\Delta_{\{i(t)\}} = 0$ or (7.16) if $\Delta_{\{i(t)\}} > 0$.
6. Tune shape parameter for $K(:, i(t))$: $\sigma_{i(t)} = \arg \min ||r||_2$.
7. Update $Q(t-1)$ and $R(t-1)$ to $Q(t)$ and $R(t)$ with $K(:, i(t))$.
8. Update residual: $r(t) \leftarrow r(t-1) - \left( Q(:, t) \right)^T y Q(:, t)$.
9. Calculate stopping criteria.
until Stopping criteria are met.

10. Solve for weights from the system: $R(t) \alpha(t) = \left( Q(t) \right)^T y$.

when normalizing the training dataset, or normalization must be skipped entirely. The reason for this is the inherent relationship between gradient training examples where $\Delta > 0$ and so-called regular examples where $\Delta = 0$. Normalizing all $y$ values in $\mathcal{T}$ to either zero mean and unit variance or to the $[0,1]$ domain will inevitably modify the input-output relationship described in the original dataset. Similarly, it is not enough to normalize only $y$ values for which $\Delta = 0$ as the gradient values no longer hold the same information for the new domain. The gradient values must also be scaled accordingly. Consider a simple example where $\Delta = 0$ for the first $m$ examples in $\mathcal{T}$ and $\Delta > 0$ for the remaining $g$ examples. A simple normalization procedure of only the $y$ values would begin by scaling the first $m$ values to zero mean and unit variance through:

$$y_i^* = \frac{y_i - \mu_y}{\sigma_y} \text{ for } i = 1, \ldots, m$$

(7.19)
where $\mu_y$ and $\sigma_y$ are the mean and standard deviation of all $y$ values for which $\Delta = 0$. The gradient values ($y$ for which $\Delta > 0$) can then be scaled through the following:

$$
y'_i = \frac{y_i}{\sigma_y} \text{ for } i = m + 1, \ldots, g.
$$

(7.20)

It must also be noted that the gradient values are again invalid if the dimensions of $x$ are normalized. An example of correct normalization would be to compute $\mu_x$ and $\sigma_x$ only for the examples where $\Delta = 0$ and transforming all examples in $T$ with those values. In addition, the corresponding gradient values must now be modified by multiplying them by $\sigma_x$. Alternatively, the training dataset may be left un-normalized. This however may lead to poor performance not just for our greedy algorithms, but most machine learning techniques.

The second hurdle is important to keep in mind when implementing these greedy algorithms because many of the significant decisions (candidate selection, shape parameter tuning, and residual convergence) are based on the residual vector, $r \in \mathbb{R}^M$. Since $r$ contains elements corresponding both to regular function values (for which $\Delta = 0$) and gradient values (for which $\Delta > 0$), a “preference factor” $\beta \in [0, 1]$ may be necessary to define in order to shift weight from one type of residual value to the other. For example, gradient values may be orders of magnitude larger than regular function values and as a result the centres for which gradient data is available will be picked more frequently in the GRBFH algorithm. If the elements of $r$ corresponding to $T$ records for which $\Delta > 0$ are weighted with $\beta = 0.1$, for example, the focus will shift more towards choosing centres associated with regular function values. Similarly, shape parameters may be tuned in a way which over- or under-weights the residual of gradient values. The right solution to this issue will ultimately depend on the approximation problem at hand.

### 7.3.4 Complexity Analysis

Table 7.1 summarizes the complexity of the individual stages for each of the algorithms considered thus far. Hermite versions of GGDL-CV and GGDL-R are excluded but their complexity is the same as their non-Hermite counterparts, with the exception of $m$ being replaced by $M = m + g$. In addition, we introduce $s$ as the number of candidate parameters considered during the tuning stage.

It is immediately evident that each of the algorithms has its own complexity strengths. GRBF, for example, benefits from having virtually no cost associated with initialization or candidate selection. GGDL on the other hand has no parameter tuning stage.
Table 7.1: Complexity analysis of GRBF, GGDL-CV, GGDL-R, GRBFH-P, and GRBFH-A. During parameter tuning, $s$ is used to designate the number of candidate parameters considered. Hermite versions of GGDL are left out, but have complexity similar to GGDL-CV and GGDL-R, where $m$ is replaced with $M = m + g$.

<table>
<thead>
<tr>
<th></th>
<th>GRBF</th>
<th>GGDL-CV</th>
<th>GGDL-R</th>
<th>GRBFH-P</th>
<th>GRBFH-A</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialization</td>
<td></td>
<td>$\mathcal{O}(md)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>candidate selection</td>
<td></td>
<td>$\mathcal{O}(mtd)$</td>
<td>$\mathcal{O}(60nm) + \mathcal{O}(60m)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>parameter tuning</td>
<td>$\mathcal{O}(stm)$</td>
<td>$\mathcal{O}(stM)$</td>
<td>$\mathcal{O}(stM)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>model update</td>
<td>$\mathcal{O}(mt)$</td>
<td>$\mathcal{O}(mt)$</td>
<td>$\mathcal{O}(mt)$</td>
<td>$\mathcal{O}(tM)$</td>
<td>$\mathcal{O}(tM)$</td>
</tr>
</tbody>
</table>

### 7.4 Numerical Examples

In this section, we will demonstrate the performance of our implementation of GRBFH-P and GRBFH-A while comparing the two algorithms to our previous GRBF technique.

We first illustrate the performance of the two GRBF techniques by approximating a dataset sampled from a 1-dimensional synthetic test function. A set of $m = 50$ examples is randomly sampled on the domain $x \in [-25, 25]$. We evaluate both the test function and its derivative at the $m = 50$ examples to produce a training set of size $M = m + g = 50 + 50 = 100$. The test function used for this example is the sinc function:

$$y = \text{sinc}(x) = \frac{\sin(x)}{x}, \quad y' = \frac{\cos(x)}{x} - \frac{\sin(x)}{x^2}. \quad (7.21)$$

Figure 7.1 shows the randomly-sampled dataset. The function values for the $m$ values of sinc are shown as dots and the $g$ gradients are shown as tangent lines through those points. We note that in an effort to best illustrate the larger gradient values, tangent lines have been extended proportionately to their absolute gradient values. Nonetheless, the dataset contains gradients at each of the $m$ points.

Because the dataset is not noisy, we expect that GRBF and the two gradient-enhanced algorithms will all perform relatively well once the stopping criteria is met. As a result, we fix the model sparsity to $t = 20$ and observe the quality of the approximation. We begin by approximating the $m = 50$ sinc examples using GRBF. We use Gaussian RBFs and tune shape parameters using a grid search on the domain $\sigma \in [0.1, 50]$. Similarly, GRBFH-P uses the same basis functions and shape parameter tuning method. It is, however, given all $M$ training set examples and is able to tune shape parameters and solve for weights in an attempt to reduce the residual for all $M$ examples. Finally, GRBFH-A is also able to add Hermite basis functions at the $m$ different centres. For this example, the Hermite basis functions are the first derivative of the Gaussian RBF,
Figure 7.1: A set of $m = 50$ points is randomly sampled in the domain $x \in [-25, 25]$ and evaluated using the $y = \text{sinc}(x)$ function and its derivative.

as in (7.14). Like the regular Gaussian RBF, the Gaussian derivative RBF is also based on a shape parameter. We tune this parameter in the same way: by grid searching on $\sigma \in [0.1, 50]$.

Figure 7.2 shows the three approximations developed at $t = 20$ with the three different algorithms. We note that GRBF, oblivious to the gradient information in the dataset, performs slightly worse than GRBFH-P, especially in the two valleys of the true function (around $x = -5$ and $x = 5$) where there is a lack of training data. Of the three approximations, GRBFH-A appears to be the best. After 5 iterations, the predictor is composed of three regular RBFs and two Hermite RBFs. It is clearly more capable of capturing the trends of the true function in the two valleys, making use of the gradient information in the nearby examples.

We would now like to compare the algorithms in their ability to drive down the training residual. Because GRBF has an “incomplete” training dataset which does not include the gradient information, it is excluded in this comparison. Figure 7.3 shows the training error trends for GRBFH-P and GRBFH-A. For the first three iterations, the two algorithms perform identically, adding RBFs at the same locations with the same shape parameters. At iteration four, however, GRBFH-A begins to set itself apart by including Hermite RBFs in its approximation.

We would also like to compare all three algorithms in their ability to approximate the true function. We use a test set of 1000 examples spaced linearly on $x \in [-25, 25]$ and calculate a test set error between the exact sinc function on this interval and the predictors constructed by each algorithm from $t = 1$ to $t = 40$. Figure 7.4 shows the test set error trends. GRBF is significantly outperformed by the two gradient-enhanced
Figure 7.2: The synthetic dataset from Figure 7.1 is approximated using the GRBF (a), GRBFH-P (b), and GRBFH-A (c) algorithms. Each algorithm is stopped after a total of $t = 20$ iterations. The GRBF algorithm uses only the function value data in the dataset, while the two other algorithms also use the gradient information.
algorithms. GRBFH-A in particular performs exceptionally well for the problem at hand, reaching a test set error of 0.05 within 25 iterations. Because the mechanics are so similar between the three algorithms, this comparison speaks to the power of including gradient information in the dataset and using an algorithm which can take advantage of that gradient data to improve the approximation.

Finally, we consider three of the test functions used in Chapter 4 for our GRBF numer-
Chapter 7. Greedy Algorithms for Gradient-Enhanced Data

123

atical results. We differentiate functions $F_1$, $F_2$, $F_3$, and $F_5$ (see Appendix A) to construct gradient-enhanced datasets and compare GRBF to both GRBFH-P and GRBFH-A. Table 7.2 contains the test error results for different training set sizes. The GRBF algorithm approximates only the $m$ function value points, while both of the Hermite algorithms are able to use the additional $g$ gradient points. All points are sampled randomly within the test function’s recommended domain and there is no requirement for derivatives to coincide with function value points.

In general, our studies found that the Hermite methods offer a substantial improvement in accuracy over the regular GRBF algorithm. Between the two Hermite techniques, the active method performed better than the passive method. It may appear that the accuracy benefits of the Hermite techniques come at the cost of sparsity, as the regular GRBF technique consistently provided predictors with fewer basis functions. However, the final predictor sparsities are highly dependent on the stopping criteria used in the algorithm. In this case, we used only two stopping criteria: a lower bound on the training residual and a lower bound on the reciprocal condition number. We believe that the Hermite algorithms take more iterations to drive down the training residual (the more “activated” criterion, from our experiments) because of the varying scales of a residual vector which consists of both function values and derivatives. We still believe that the Hermite algorithms offer more accurate predictors at similar sparsities, as shown earlier in Figure 7.4.

7.5 Summary and Concluding Remarks

In this chapter, we presented two “Hermite” methods for adapting our greedy algorithms to problems with gradient-enhanced data. We began by discussing the motivation behind these adaptations, namely problems for which sensitivity analysis may be used to obtain gradient data. We then showed that the popular Hermite RBF interpolation approach suffers from the same setbacks as classical RBF approximation – difficulties in tuning the global shape parameter and high cost. After demonstrating our approach to including gradient data in our greedy learning setting, we worked to adapt our GRBF approach from Chapter 3 to problems where gradient data is available.

Our first “passive” method (GRBFH-P) uses the gradient data examples to improve accuracy during the shape parameter tuning and model update stages. Our second “active” method (GRBFH-A) builds upon GRBFH-P by also adding derivative basis functions to the approximation. We noted that extending these methods to our two GGDL algorithms from Chapter 5 is straightforward and requires only that the dictio-
Table 7.2: The accuracy of three RBF algorithms is compared for gradient-enhanced datasets. Test functions are used to create gradient-enhanced datasets with $m$ function values points and $g$ gradient points. All points are sampled randomly within the function’s recommended domain (see Appendix A). The GRBF, GRBFH-P, and GRBFH-A algorithms are used to approximate the data. Test error is calculated using a 1000-point dataset. Predictor sparsities are given in brackets. For the $n = 2$ dimensional $F_5$, $g$ gradient points are used for each dimension.

<table>
<thead>
<tr>
<th>$m$</th>
<th>$g$</th>
<th>$F_1$</th>
<th>$F_2$</th>
<th>$F_3$</th>
<th>$F_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25</td>
<td>2.462E-1 (9)</td>
<td>9.817E-2 (32)</td>
<td>8.638E-4 (38)</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>3.242E-3 (39)</td>
<td>2.973E-3 (42)</td>
<td>3.332E-4 (37)</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7.702E-4 (35)</td>
<td>6.044E-5 (40)</td>
<td>3.192E-5 (47)</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>4.931E-1 (12)</td>
<td>5.780E-1 (33)</td>
<td>5.050E-2 (42)</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>1.934E-1 (9)</td>
<td>4.109E-5 (50)</td>
<td>6.995E-6 (53)</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>1.791E-2 (49)</td>
<td>9.370E-6 (57)</td>
<td>3.522E-6 (56)</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>2.653E-5 (19)</td>
<td>3.460E-6 (24)</td>
<td>5.882E-6 (25)</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>1.353E-5 (23)</td>
<td>4.493E-6 (25)</td>
<td>4.556E-7 (23)</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>2.142E-6 (24)</td>
<td>3.343E-7 (26)</td>
<td>2.371E-8 (25)</td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>100</td>
<td>9.748E-1 (30)</td>
<td>9.451E-1 (6)</td>
<td>9.274E-1 (6)</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>200</td>
<td>5.931E-1 (329)</td>
<td>7.095E-1 (354)</td>
<td>4.040E-1 (324)</td>
<td></td>
</tr>
<tr>
<td>750</td>
<td>300</td>
<td>8.042E-1 (353)</td>
<td>2.109E-1 (370)</td>
<td>4.890E-1 (412)</td>
<td></td>
</tr>
</tbody>
</table>

Numerical studies on a synthetic dataset showed that surrogate models built with gradient data can improve on those built using only function value data in both accuracy and sparsity. In addition, we showed that the active method can outperform the passive method in decreasing both training and test set errors with fewer basis functions.
Chapter 8

Conclusions and Future Work

8.1 Summary & Contributions

This thesis began with a thorough literature review (Chapter 2) which covered techniques for scattered data approximation as well as the motivation and techniques for sparse dictionary learning. Novel contributions to the field began in Chapter 3.

A greedy radial basis function (GBRF) approximation algorithm was introduced in Chapter 3 as a novel way to learn using parameterized dictionaries. The algorithm was motivated by two observed setbacks associated with classical RBF approximations: difficulties tuning the global shape parameter value, and high computation and memory costs. An efficient thin QR update scheme was employed to handle column-wise growth of the linear algebraic system at each iteration. The algorithm made use of a collection of stopping criteria, including a check for an increase in the moving window average of an approximate leave-one-out error. Numerical results for the GRBF algorithm were documented in Chapter 4. These consisted of illustrative examples, such as a visual evolution of the predictor and iteration-wise trends in training error and condition number, as well as accuracy, cost, and sparsity results for a number of test functions. Test function results were compared to classical RBF approximations and Gaussian process models (GPMs). In general, the GRBF algorithm was capable of constructing approximations with levels of accuracy and cost which are comparable to these two state-of-the-art techniques for function approximation. At the same time, GRBF’s predictors were significantly more sparse. We concluded that the GRBF algorithm should be a serious candidate for scattered data approximation tasks where predictor sparsity is of importance.

A greedy algorithm for general dictionary learning (GGDL) was introduced in Chapter 5 as a novel way to learn using a heterogeneous dictionary of basis functions and construct incredibly-flexible surrogates. An efficient QR update scheme was once again employed
Chapter 8. Conclusions and Future Work

and two variations of GGDL were described. GGDL-CV implemented a novel parallel strategy and relied on a cross-validation based stopping. GGDL-R employed a randomized selection criteria to significantly reduce the cost associated with large datasets and large dictionaries. Both variations of GGDL also employed an efficient basis function selection criterion based on an expression for the future training residual. Numerical results for GGDL were documented in Chapter 6. Similar to the results for GRBF, these first began with a collection of illustrative examples which demonstrated predictor evolution for GGDL-CV and GGDL-R. Iteration-wise trends in approximation metrics such as training error were also provided for a couple of synthetic datasets. GGDL-CV and GGDL-R were then used to approximate four popular machine learning datasets – Abalone, Boston, CPU Small, and Diabetes. Results for these datasets were compared to both support vector machines (SVMs) and Gaussian process models (GPMs). GGDL-CV was typically able to build predictors with significant sparsity benefits and accuracy which was comparable to the two competing methods. At the same time, GGDL-R built predictors with only slightly more basis functions (still significantly fewer than SVMs or GPMs), but in less time and often with improved accuracy. In general, the GGDL algorithm was a very strong contender and we recommended considering both GGDL-CV and GGDL-R for a wide range of approximation problems.

Chapter 6 concluded with two case studies where we demonstrated how the GGDL algorithm may be applied to some practical problems. GGDL-CV was first used to construct sparse polynomial chaos (PC) expansions for an uncertainty quantification problem. Our application of the GGDL algorithm showed that sparse PC expansions can be constructed with higher order polynomials. We used a dataset from the thermal driven cavity flow problem with stochastic boundary conditions to compare GGDL-CV to both Quasi Monte Carlo approximations and full PC expansions and showed improvements in both accuracy and sparsity. GGDL was then applied to a gappy temporal data problem to discuss a number of ways in which the algorithm may tackle scattered data problems of this nature. Gappy particle image velocimetry (PIV) data from the field of gas turbine combustor design was used for this case study.

Finally, in Chapter 7 we presented two “Hermite” methods for adapting our greedy algorithms to problems where the dataset contains gradient-enhanced data. Our first “passive” method made use of the gradient-enhanced data to improve predictor accuracy during algorithms steps such as shape parameter tuning. Our second “active” method went further by also using derivative basis functions in the basis function dictionary. The chapter concluded with numerical studies on a synthetic dataset and a collection of test functions which demonstrated that gradient-enhanced datasets have the potential to
significantly improve predictor accuracy and that of the two Hermite methods, the active method performs best in decreasing both training and test set errors using a smaller number of basis functions.

8.2 Future Work

The work presented in this thesis covers a number of contributions to the field of surrogate modelling. At the same time, this work opens a number of doors for future research. Below we list some open problems in the form of research questions. A number of these problems have been touched upon in this thesis, while the ideas behind others are more tangential to this thesis work.

- The focus of this thesis was primarily on the subclass of supervised machine learning problems known as “regression”. This raises a number of interesting research questions about the effectiveness of the GRBF and GGDL algorithms for classification problems. In addition, research may look into ways for modifying the algorithms in a way that would improve classification performance?

- The GGDL studies presented in Chapter 6 dealt largely with a specific choice of basis function dictionary $\mathcal{D}$. Further research is required to observe the effects of “dictionary engineering” (strategic selection of basis function centres, kernel types, and shape parameters) on algorithm performance. It is important to consider whether there are problems which benefit significantly from careful dictionary engineering and whether we can design ways to automate the “construction” of basis function dictionaries in the GGDL algorithm while maintaining high levels of accuracy.

- In Chapter 7 we presented a way to approximate the most general of gradient-enhanced datasets with our greedy algorithms, but did not conduct a significant collection of numerical studies to assess the potential impact of these methods. Can substantial case studies help us determine the impact of the Hermite variations of our GRBF and GGDL algorithms?

- After observing significant cost reductions from using a randomized basis function candidate sampling approach (Section 5.4.1), it is hypothesized that this technique may yield additional improvements if applied to the rows of the regressor matrix. Research may look into applying the random subsampling approach from Section 5.4.1 to compute basis function values for only a representative sample of the full dataset.
• It is expected that the randomized candidate selection criterion introduced in the GGDL-R algorithm can be used to construct sparse approximate preconditioners (Grote & Huckle, 1997; Gould & Scott, 1998) and lead to a speed up factor of as high as $\frac{m}{\ln m}$, where $m$ is the leading dimension of the target matrix. Future work may look into developing an algorithm for constructing sparse approximate preconditioners using randomization.

• Recent work in Gaussian process modelling has suggested that Kronecker matrix algebra may be used to significantly reduce computational cost (Wilson et al., 2014). Future research may look into the application of Kronecker matrix algebra to greedy algorithms.
Appendix A – Test Functions

Function $F_1$ is a one-dimensional function obtained from Xiong et al. (2007). It is meant to be sampled on the interval $[0, 1]$.

$$F_1(x) = \sin(30(x - 0.9)^4) \cos(2(x - 0.9)) + \frac{x - 0.9}{2}$$

Function $F_2$ is a one-dimensional function obtained from Gramacy & Lee (2012). It is meant to be sampled on the interval $[0.5, 2.5]$.

$$F_2(x) = \frac{\sin(10\pi x)}{2x} + (x - 1)^4$$

Function $F_3$ is a one-dimensional function obtained from Forrester et al. (2008). It is meant to be sampled on the interval $[0, 1]$.

$$F_3(x) = (6x)^2 \sin(12x - 4)$$

Function $F_4$ is a two-dimensional function obtained from Ritchie (1978). It is meant to be sampled on the interval $[0, 1]$ for all $x_i$.

$$F_4(x_1, x_2) = \begin{cases} 
1 & : (x_2 - \xi) \geq 0.5 \\
2(x_2 - \xi) & : 0 \leq (x_2 - \xi) \leq 0.5 \\
(cos(4\pi r(\xi, x_2)) + 1)/2 & : r(\xi, x_2) \leq 0.25 \\
0 & : otherwise 
\end{cases}$$

where

$$\xi = 2.1x_1 - 0.1$$

$$r(\xi, x_2) = \sqrt{\left(\frac{\xi}{2} - \frac{3}{2}\right)^2 + \left(x_2 - \frac{1}{2}\right)^2}$$

Function $F_5$ is a two-dimensional function obtained from Friedman (1999). It is meant
to be sampled on the interval $[0, 1]$ for all $x_i$.

$$F_5(x_1, x_2) = \sin(30(x_2 - 0.9)^4) \cos(2(x_1 - 0.9)) + \frac{x_2 - 0.9}{2}$$

Function $F_6$ is a two-dimensional function obtained from Pires et al. (2010). It is meant to be sampled on the interval $[-5.12, 5.12]$ for all $x_i$.

$$F_6(x_1, x_2) = -\frac{1 + \cos(12\sqrt{x_1^2 + x_2^2})}{\frac{1}{2}(x_1^2 + x_2^2) + 2}$$

Function $F_7$ is an n-dimensional function obtained from Laguna & Marti (2005). It is meant to be sampled on the interval $[-500, 500]$ for all $x_i$.

$$F_7(x_1, \ldots, x_n) = 418.9829n - \sum_{i=1}^{n} x_i \sin(\sqrt{|x_i|})$$

Function $F_8$ is an n-dimensional function obtained from Back (1996). It is meant to be sampled on the interval $[-32.768, 32.768]$ for all $x_i$, with the recommended values $a = 20$, $b = 0.2$, $c = 2\pi$.

$$F_8(x_1, \ldots, x_n) = -a \exp \left(-b \sqrt{\frac{1}{n} \sum_{i=1}^{n} x_i^2}\right) - \exp \left(\frac{1}{n} \sum_{i=1}^{n} \cos(c x_i)\right) + a + \exp(1)$$

Function $F_9$ is an n-dimensional function obtained from Laguna & Marti (2005). It is meant to be sampled on the interval $[-10, 10]$ for all $x_i$.

$$F_9(x_1, \ldots, x_n) = \sin^2(\pi w_1) + \sum_{i=1}^{n-1} (w_i - 1)^2 \left[ 1 + 10 \sin^2(\pi w_i + 1) \right] + (w_n - 1)^2 \left[ 1 + \sin^2(2\pi w_n) \right]$$

where $w_i = 1 + \frac{x_i - 1}{4}$.
Appendix B – Datasets

Abalone

The Abalone dataset was obtained from the UCI Machine Learning Repository [Lichman, 2013]. The 8-dimensional dataset consists of 4177 examples. The 8 attributes are physical measurements of abalone and the output is the age of the abalone. Original data comes from a study by Nash et al. [1994].

Boston Housing

The Boston Housing dataset was obtained directly from the open source Python machine learning library Scikit-Learn [Pedregosa et al., 2011]. The load_boston routine within Scikit-Learn’s datasets module makes it easy to perform experiments on this dataset. The 13-dimensional dataset consists of 506 examples. The 13 attributes are information about housing in a Boston suburb and the output is the median value of an home in that suburb. Original data is from a study by Harrison & Rubinfeld [1978].

CPU Small

The CPU Small dataset, also known as the Comp-Activ dataset, was obtained from the LIBSVM database [Chang & Lin, 2011]. The 27-dimensional dataset consists of 8192 examples. The 27 attributes are various computer system performance measures and the output is a measure of computer system activity.

Diabetes

The Diabates dataset was obtained directly from the open source Python machine learning library Scikit-Learn [Pedregosa et al., 2011]. The load_diabetes routine within Scikit-Learn’s datasets module makes it easy to perform experiments on this dataset. The 10-dimensional dataset consists of 442 examples. The 10 attributes are various pieces of information about a patient and the output is a quantitative measure of disease progression one year after baseline. Original data is from a study by Efron et al. [2004].
Bibliography


