ABSTRACT

Title of dissertation: SCALABLE MACHINE LEARNING FOR MASSIVE DATASETS : FAST SUMMATION ALGORITHMS

Vikas Chandrakant Raykar
Doctor of Philosophy, 2007

Dissertation directed by: Professor Dr. Ramani Duraiswami
Department of Computer Science

Huge data sets containing millions of training examples with a large number of attributes are relatively easy to gather. However one of the bottlenecks for successful inference is the computational complexity of machine learning algorithms. Most state-of-the-art nonparametric machine learning algorithms have a computational complexity of either $\mathcal{O}(N^2)$ or $\mathcal{O}(N^3)$, where N is the number of training examples. This has seriously restricted the use of massive data sets. The bottleneck computational primitive at the heart of various algorithms is the multiplication of a structured matrix with a vector, which we refer to as *matrix-vector product* (MVP) primitive. The goal of my thesis is to speedup up some of these MVP primitives by *fast approximate algorithms* that scale as $\mathcal{O}(N)$ and also provide *high accuracy guarantees*. I use ideas from computational physics, scientific computing, and computational geometry to design these algorithms. The proposed algorithms have been applied to speedup kernel density estimation, optimal bandwidth estimation, projection pursuit, Gaussian process regression, implicit surface fitting, and ranking.

SCALABLE MACHINE LEARNING FOR MASSIVE DATASETS: FAST SUMMATION ALGORITHMS

by

Vikas Chandrakant Raykar

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2007

Advisory Committee:
Dr. Ramani Duraiswami, Chair/Advisor
Dr. Min Wu, Dean's Representative
Dr. Nail Gumerov
Dr. Lise Getoor
Dr. Howard Elman

© Copyright by
Vikas Chandrakant Raykar
2007

Acknowledgements

First and foremost I would like to thank my advisor Dr. Ramani Duraiswami. Ramani has been more of a mentor and a friend. Interacting with him was as easy and informal as talking to any graduate student in the department. His infectious enthusiasm for new ideas helped me work on a diverse topics during my five years at Maryland. Ramani is quite an expert on fast multipole methods, which is a main source of inspiration for the algorithms developed in this thesis.

I would also like to thank Dr. Nail Gumerov for the discussions I had with him. He has this uncanny ability to attack any problem from the first principles. A part of this thesis has been a improvement of the work done by Dr. Changjiang Yang. I would like to thank him for some of the code which I used in my thesis. I would also like to thank Nargess Memarsadeghi (soon to be Dr.) for helping me with the ANN library and also for pointing out the many bugs in the FIGTree code. I collaborated with her for some work on kriging. Also I would like to thank my lab mates and colleagues Dr. Zhiyun Li, Dr. Dmitry Zotkin, Dr. Elena Grassi, Dr. Vinay Shet, Shiv Vittaldevuni, and Son Tran. I learnt quite a lot from them. Most of the code I developed during the thesis has been released as open source software. I would like to thank all those who downloaded the code and pointed out the bugs. Finally I would like to thank Dr. Howard Elman, Dr. Lise Getoor, and Dr. Min Wu for serving on my dissertation committee and providing comments on the manuscript.

I would also like to thank Dr. Balaji Krishnapuram from Siemens Medical Solutions. The work on ranking was done in collaboration with him. Balaji is a

walking bibliography on anything related to machine learning and statistics. His problem formulation skills are enviable. I was also fortunate to have worked with Dr. Harald Steck during my internship at Siemens Medical Solutions. I learnt from him on how to patiently think about problems from first principles and develop deep insights. I did some nice work on survival analysis with him–which however is not a part of this thesis. I would also like to thanks my co-interns Indri and Meliha, who made the internship a much nicer experience.

A special note of thanks goes to my roommates and friends–arun, murali, niranjan, aamer, vishwa, punya, and punarbasu. Thanks are due to my wonderful dancing parter Jane. The practice sessions we had during the semester really helped to to clear my mind during the thesis writing sessions.

Last but not the least I am forever indebted for my parents, for I am what I am wholly due to their efforts and sacrifices.

Table of Contents

List of Tables viii

List of Figures x

1 Introduction 1
1.1 Computational curse of non-parametric methods 1
1.2 Bringing computational tractability to massive datasets 2
1.3 Weighted superposition of kernels 3
1.4 Fast approximate matrix-vector product 5
1.5 Fast multipole methods 6
1.6 Motivating example–polynomial kernel 7
1.7 Thesis contributions 9

2 Algorithm 1: Fast weighted summation of multivariate Gaussians 11
2.1 Discrete Gauss transform 12
2.2 Related work 13
2.2.1 Methods based on sparse data-set representation 13
2.2.2 Binned Approximation based on the FFT 14
2.2.3 Dual-tree methods 14
2.2.4 Fast Gauss transform 15
2.3 Improved fast Gauss transform 16
2.4 Preliminaries 18
2.4.1 Multidimensional Taylor Series 18
2.4.2 Multi-index Notation 22
2.4.3 Space sub-division 23
2.5 Improved Fast Gauss Transform 26
2.5.1 Factorization 27
2.5.2 Regrouping 28
2.5.3 Space subdivision 28
2.5.4 Rapid decay of the Gaussian 29
2.5.5 Runtime analysis 29
2.5.6 Storage analysis 31
2.5.7 Efficient computation of multivariate polynomials–Horner's rule 31
2.5.8 Partial distance 32
2.6 Choosing the parameters 33
2.6.1 Automatically choosing the cut off radius for each cluster . . . 35
2.6.2 Automatically choosing the truncation number for each source 36
2.6.3 Automatically choosing the number of clusters 38
2.6.4 Updating the truncation number 40
2.7 Nearest Neighbor search based on *kd*-tree 42
2.8 FIGTree–Fast Improved Gauss Transform with *kd*-Tree 43
2.9 Comparison with the Fast Gauss Transform 44
2.9.1 Comparison of the IFGT and FGT factorizations 45

2.9.2 Translation 47
2.9.3 Error bounds 48
2.9.4 Spatial data structures 49
2.9.5 Exponential growth of complexity with dimension 50
2.10 Numerical Experiments 51
2.10.1 Speedup as a function of N 52
2.10.2 Speedup as a function of d 54
2.10.3 Speedup as a function of the bandwidth h 59
2.10.4 Speedup as a function of the desired error ϵ 63
2.10.5 Structured data 63
2.10.6 Summary 66
2.11 Applications 68
2.12 Conclusions 69

3 Algorithm 2: Fast weighted summation of univariate Hermite × Gaussians 75
3.1 Introduction 76
3.2 Factorization of the Gaussian 77
3.3 Factorization of the Hermite polynomial 79
3.4 Regrouping of the terms 80
3.5 Space subdivision 80
3.6 Decay of the Gaussian 81
3.7 Computational and space complexity 82
3.8 Error bounds and choosing the parameters 82
3.8.1 Choosing the cut-off radius 83
3.8.2 Choosing the truncation number 84
3.8.3 Choosing the interval length 85
3.9 Numerical experiments 87

4 Algorithm 3: Fast weighted summation of erfc functions 93
4.1 Introduction 94
4.2 Series expansion 95
4.3 Fast summation algorithm 98
4.4 Runtime and storage analysis 99
4.5 Direct inclusion and exclusion of faraway points 99
4.6 Space sub-division 100
4.7 Choosing the parameters 102
4.8 Numerical experiments 104

5 Kernel density estimation 108
5.1 Kernel density estimation 109
5.2 Bandwidth selection 110
5.3 Experiments 111

6 Optimal bandwidth estimation 114
6.1 Introduction . . . 115
6.2 Kernel Density Estimation . . . 117
6.2.1 AMISE optimal bandwidth for kernel density estimate . . . 119
6.3 Kernel Density Derivative estimation . . . 121
6.3.1 AMISE optimal bandwidth for kernel density derivative estimate 122
6.4 Estimation of Density Functionals . . . 122
6.4.1 AMSE optimal bandwidth for density functional estimation . 124
6.5 AMISE optimal Bandwidth Selection . . . 124
6.5.1 Review of different methods . . . 125
6.5.2 Solve-the-equation plug-in method . . . 126
6.6 Fast $\epsilon - exact$ density derivative estimation . . . 129
6.7 Speedup achieved for bandwidth estimation . . . 130
6.7.1 Synthetic data . . . 130
6.7.2 Real data . . . 131
6.8 Projection Pursuit . . . 131
6.9 Related work . . . 137
6.10 Conclusions . . . 140

7 Gaussian process regression 141
7.1 Introduction . . . 142
7.2 Gaussian process model . . . 143
7.3 Conjugate Gradient . . . 146
7.4 Fast matrix-vector products . . . 148
7.5 The accuracy ϵ, necessary . . . 149
7.6 Prediction . . . 151
7.7 Experiemnts . . . 154
7.8 Implicit Surface fitting . . . 158
7.9 Discussion/Further issues . . . 161

8 Large scale preference learning 163
8.1 Introduction . . . 164
8.1.1 Preference relation and ranking function . . . 164
8.1.2 Problem statement . . . 166
8.1.3 Generalized Wilcoxon-Mann-Whitney statistic . . . 168
8.1.4 Our proposed approach . . . 169
8.1.5 Organization . . . 169
8.2 Previous literature on learning ranking functions . . . 170
8.2.1 Methods based on pair-wise relations . . . 170
8.2.2 Fast approximate algorithms . . . 171
8.2.3 Other approaches . . . 172
8.2.4 WMW maximizing algorithms . . . 172
8.2.5 Relationship to the current paper . . . 173
8.3 The MAP estimator for learning ranking functions . . . 174
8.3.1 Lower bounding the WMW . . . 175

8.4 The optimization algorithm 176
8.4.1 Gradient approximation using the error-function 177
8.4.2 Quadratic complexity of gradient evaluation 178
8.5 Fast weighted summation of erfc functions 180
8.6 Ranking experiments 181
8.6.1 Datasets 181
8.6.2 Evaluation procedure 181
8.6.3 Results 183
8.6.3.1 Quality of approximation 183
8.6.3.2 Comparison with other methods 184
8.6.3.3 Ability to handle large datasets 184
8.6.4 Impact of the gradient approximation: 184
8.7 Application to Collaborative filtering 189
8.8 Conclusion and future work 189
8.8.1 Future Work 191

9 Conclusions 193
9.1 Future work 193
9.2 Open problems 194

Bibliography 195

List of Tables

1.1 *Summary of the thesis.* The fast summation algorithms designed and tasks to which they were applied. Computation of each of these primitives at M points requires $\mathcal{O}(MN)$ time. The fast algorithms we design computes the same to a specified ϵ accuracy in $\mathcal{O}(M+N)$ time. 10

2.1 Number of multiplications required for the direct and the efficient method for evaluating all d-variate monomials of degree less than or equal to n. 33

2.2 Comparison of the different parameters chosen by the FGT and the IFGT as a function of the data dimensionality d. $N = 100,000$ points were uniformly distributed in a unit hyper cube. The bandwidth was $h = 0.5$ and the target error was $\epsilon = 10^{-6}$. 73

2.3 Summary of the better performing algorithms for different settings of dimensionality d and bandwidth h (assuming data is scaled to a unit hypercube). The bandwidth ranges are approximate. 74

5.1 *KDE experiments on the entire dataset.* Time taken by the direct summation and the FIGTree on the entire dataset containing $N = 44,484$ source points. The KDE was evaluated at $M = N$ points. The error was set to $\epsilon = 10^{-2}$. 113

6.1 The bandwidth estimated using the solve-the-equation plug-in method for the fifteen normal mixture densities of Marron and Wand. h_{direct} and h_{fast} are the bandwidths estimated using the direct and the fast methods respectively. The running time in seconds for the direct and the fast methods are shown. The absolute relative error is defined as $|h_{direct} - h_{fast}/h_{direct}|$. In the study $N = 50,000$ points were sampled from the corresponding densities. For the fast method we used $\epsilon = 10^{-3}$. 132

6.2 Optimal bandwidths for the five continuous attributes for the Adult database from the UCI machine learning repository. The database contains 32,561 training instances. The bandwidth was estimated using the solve-the-equation plug-in method. h_{direct} and h_{fast} are the bandwidths estimated using the direct and the fast methods respectively. The running time in seconds for the direct and the fast methods are shown. The absolute relative error is defined as $|h_{direct} - h_{fast}/h_{direct}|$. For the fast method we used $\epsilon = 10^{-3}$. 134

7.1 *The dominant computational and space complexities.* We have N training points and M test points in d dimensions. k is the number of iterations required by the conjugate gradient procedure to converge to a specified tolerance. The memory requirements are for the case where the Gram matrix is constructed on the fly at each iteration. For the fast MVM procedure, the constant $\mathcal{D}$ grows with d depending on the type of fast MVM used. 153

8.1 Benchmark datasets used in the ranking experiments. N is the size of the data set. d is the number of attributes. S is the number of classes. $\mathcal{M}$ is the average total number of pairwise relations per fold of the training set. 186

8.2 The mean training time and standard deviation in seconds for the various methods and all the datasets shown in Table 8.1. The results are shown for a five fold cross-validation experiment. The symbol $\star$ indicates that the particular method either crashed due to limited memory requirements or took a very large amount of time. 187

8.3 The corresponding generalized Wilcoxon-Mann-Whitney statistic on the test set for the results shown in Table 8.2. 188

8.4 Results for the EACHMOVIE dataset: The mean training time (averaged over 100 users) as a function of the number of features d. . . . 190

8.5 The corresponding generalized WMW statistic on the test set for the results shown in Table 8.4. 190

List of Figures

2.1 The constant term for the FGT and the IFGT complexity as a function of the dimensionality d. The FGT is exponential in d, while the IFGT is polynomial in d. 17

2.2 (a) The actual residual (solid line) and the bound (dashed line) given by Equation (2.7) as a function of x. [$x_* = 0$, $y = 1.0$, $h = 0.5$, $r_x = 0.5$, $r_y = 1.0$, and $p = 10$]. The residual is minimum at $x = x_*$ and increases as x moves away from x_*. The dotted line shows the very pessimistic bound which is independent of x (Equation (2.12)) used in the original IFGT. (b) The truncation number p required as a function of x so that the error is less than10^{-6}. 21

2.3 (a) Using the farthest-point clustering algorithm 10,000 points uniformly distributed in a unit square are divided into 22 clusters with the maximum radius of the clusters being 0.2. (b) 10,000 points normally distributed in a unit square are divided into 11 clusters with the maximum radius of the clusters being 0.2. 25

2.4 Efficient expansion of multivariate polynomials. 32

2.5 The error at y_j due to source x_i, i.e., Δ_{ij} [Equation (2.40)] as a function of $\|y_j - c_k\|$ for different values of p and for (a) $h = 0.5$ and (b) $h = 1.0$. The error increases as a function of $\|y_j - c_k\|$, reaches a maximum and then starts decreasing. The maximum is marked as '*'. $q_i = 1$ and $\|x_i - c_k\| = 0.5$. 37

2.6 The error at y_j due to a source at x_i, i.e., Δ_{ij} [Equation (2.40)] as a function of $\|x_i - c_k\|$ for different values of p, $h = 0.4$ and $\|y_j - c_k\| = \min\left(\|y_j - c_k\|_*, r_y^k\right)$. (b) The truncation number p_i required to achieve an error of $\epsilon = 10^{-3}$. 39

2.7 The constant c (see Equation (2.44)) as a function of K. $d = 2$, $h = 0.3$, and $\epsilon = 10^{-6}$. 40

2.8 (a) Schematic of the evaluation of the improved fast Gauss transform at the target point y_j. For the source points the results of the farthest point clustering algorithm are shown along with the center of each cluster. A set of coefficients are stored at the center of each cluster. Only the influential clusters within radius r of the target point are used in the final summation. (b) Illustration of the truncation number p_i required for each data point. Dark red color indicates a higher truncation number. Note that for points close to the cluster center the truncation number required is less than that at the boundary of the cluster. 41

2.9 The absolute value of the actual error between the one dimensional Gaussian ($e^{-(x_i-y)/h^2}$) and different series approximations. The Gaussian was centered at $x_i = 0$ and $h = 1.0$. All the series were expanded about $x_* = 1.0$. $p = 5$ terms were retained in the series approximation. 47

2.10 The total number of terms required by the IFGT and the FGT series expansions to achieve a desired error bound. 49

2.11 *Scaling with N for $d = 3$ and $h = 0.25$.* (a) The running times in seconds and (b) the maximum absolute error relative to the total weight Q for direct evaluation, FGT, dual tree, and FIGTree. The target error was set to 10^{-3}. The bandwidth was $h = 0.25$. The source and target points were uniformly distributed in a unit cube. The weights q_i were uniformly distributed between 0 and 1. For $N > 25600$ the timing results for the direct evaluation were obtained by evaluating the Gauss transform at $M = 100$ points and then extrapolating the results. The dual-tree algorithm could not be run after a certain N due to limited memory. 55

2.12 The theoretical complexity ($\mathcal{O}(2p^dN+dp^{d+1}(2n+1)^d min((\sqrt{2}rh)^{-d/2}, N))$) of the FGT showing different regions as a function of $N = M$ for $d = 3$. 56

2.13 *Scaling with N for $d = 3$ and $h = 0.05$.* Same as Figure 2.11 with $h = 0.05$. 57

2.14 *Scaling with N for $d = 3$ and $h = 1.0$.* Same as Figure 2.11 with $h = 1.0$. 58

2.15 *Scaling with d for $h = 1.0$.* (a) The running times in seconds and (b) the maximum absolute error relative to the total weight Q for direct evaluation, FGT, and FIGTree as a function of the dimension d. The target error was set at $\epsilon = 10^{-3}$. The bandwidth was $h = 1.0$. $N = 50,000$ source and target points were uniformly distributed in a unit hyper cube. The weights q_i were uniformly distributed between 0 and 1. 60

2.16 *Effect of bandwidth $h = 0.5\sqrt{d}$.*(a) The running times in seconds and (b) the maximum absolute error relative to the total weight Q for direct evaluation, FGT, and FIGTRee as a function of the dimension d. The target error was set at $\epsilon = 10^{-3}$. The bandwidth was $h = 0.5\sqrt{d}$. $N = 20,000$ source and target points were uniformly distributed in a unit hyper cube. The weights q_i were uniformly distributed between 0 and 1. 61

2.17 *Scaling with d for $h = 0.001$.* (a) The running times in seconds and (b) the maximum absolute error relative to the total weight Q for direct evaluation, FGT, and FIGTRee as a function of the dimension d. The target error was set at $\epsilon = 10^{-3}$. The bandwidth was $h = 0.001$. $N = 10,000$ source and target points were uniformly distributed in a unit hyper cube. The weights q_i were uniformly distributed between 0 and 1. 62

2.18 *Effect of bandwidth h.* (a) The running times in seconds and (b) the maximum absolute error relative to the total weight Q for direct evaluation, IFGT, and the kd-tree dual-tree algorithm as a function of the bandwidth h. The target error was set at $\epsilon = 10^{-3}$. $N = 7,000$ source and target points were uniformly distributed in a unit hyper cube of dimension $d = 2, 3, 4, \text{and}, 5$. The weights q_i were uniformly distributed between 0 and 1. 64

2.19 *Effect of the desired error ϵ.* (a) The running times in seconds and (b) the maximum absolute error relative to the total weight Q for the dual-tree and the FIGTree as a function of the desired error ϵ. The bandwidth was $h = 0.03$. $N = 5,000$ source and target points were uniformly distributed in a unit hyper cube of dimension $d = 2$. The weights q_i were uniformly distributed between 0 and 1. 65

2.20 *Effect of clumpy data.* The running times for the different methods as a function of the bandwidth h. [$\epsilon = 10^{-3}$, $N = M = 7,000$, and $d = 4$] 67

3.1 The error at y_j due to source x_i, i.e., Δ_{ij} [Eq. 3.28] as a function of $\|y_j - c_n\|$ for different values of p and for $h = 0.1$ and $r = 4$. The error increases as a function of $\|y_j - c_n\|$, reaches a maximum and then starts decreasing. The maximum is marked as '*'. $q_i = 1$ and $\|x_i - c_n\| = 0.1$. 86

3.2 (a) The running time in seconds and (b) maximum absolute error relative to Q for the direct and the fast methods as a function of N. $N = M$ source and the target points were uniformly distributed in the unit interval. For $N > 25,600$ the timing results for the direct evaluation were obtained by evaluating the result at $M = 100$ points and then extrapolating. [$h = 0.1$, $r = 4$, and $\epsilon = 10^{-6}$.] 89

3.3 (a) The speedup achieved and (b) maximum absolute error relative to Q for the direct and the fast methods as a function of ϵ. $N = M = 50,000$ source and the target points were uniformly distributed in the unit interval. [$h = 0.1$ and $r = 4$] 90

3.4 (a) The running time in seconds and (b) maximum absolute error relative to Q for the direct and the fast methods as a function of h. $N = M = 50,000$ source and the target points were uniformly distributed in the unit interval. [$\epsilon = 10^{-6}$ and $r = 4$] 91

3.5 (a) The running time in seconds and (b) maximum absolute error relative to Q for the direct and the fast methods as a function of r. $N = M = 50,000$ source and the target points were uniformly distributed in the unit interval. [$\epsilon = 10^{-6}$ and $h = 0.1$] 92

4.1 The erfc function. 94

4.2 (a) The maximum absolute error between the actual value of erfc and the truncated series representation (Eq. 7.6) as a function of the truncation number p for any $x \in [-4, 4]$. The error bound (Eq. 4.6) is also shown. (b) A sample plot of the actual erfc function and the $p = 3$ truncated series representation. The error as a function of x is also shown in the lower panel. 96

4.3 (a) The erfc function (b) The value of r for which $\mathrm{erfc}(z) < \epsilon$, $\forall z > r$ 101

4.4 (a) The running time in seconds and (b) maximum absolute error relative to Q_{abs} for the direct and the fast methods as a function of $N = M$. For $N > 3,200$ the timing results for the direct evaluation were obtained by evaluating the sum at $M = 100$ points and then extrapolating (shown as dotted line). 105

4.5 (a) The speedup achieved and (b) maximum absolute error relative to Q for the direct and the fast methods as a function of ϵ for $N = M = 3,000$. 106

5.1 *KDE experiments for $N = 7,000$ source points.* The run time in seconds for the direct, FIGTree, and the dual-tree method for varying dimensionality, d. The results are shown for a range of bandwidths around the optimal bandwidth marked by the straight line in each of the plots. The error was set to $\epsilon = 10^{-2}$. The KDE was evaluated at $M = N$ points. 112

6.1 The fifteen normal mixture densities of Marron and Wand. The solid line corresponds to the actual density while the dotted line is the estimated density using the optimal bandwidth estimated using the fast method. 133

6.2 The estimated density using the optimal bandwidth estimated using the fast method, for two of the continuous attributes in the Adult database from the UCI machine learning repository. 134

6.3 (a) The original image. (b) The centered and scaled RGB space. Each pixel in the image is a point in the RGB space. (c) KDE of the projection of the pixels on the most interesting direction found by projection pursuit. (d) The assignment of the pixels to the three modes in the KDE. 138

7.1 (a) The mean prediction and the error bars obtained when a Gaussian process was used to model the data shown by the black points. A squared exponential covariance function was used. Note that the error bars increase when there is sparse data. The hyperparameters h and σ we chosen by minimizing the negative log-likelihood of the training data the contours of which are shown in (b). 144

7.2 The error for the IFGT ϵ_k selected at each iteration for a sample 1D regression problem. The error tolerance for the CG was set to $\eta = 10^{-3}$ and $\delta = 10^{-3}$. 152

7.3 (a) The total training time, (b) the SMSE, and (c) the testing time as a function of m for the robotarm dataset. The errorbars show one standard deviation over a 10-fold experiment. The results are slightly displaced w.r.t. the horizontal axis for clarity. The lower panel show the same for the abalone dataset. 156

7.4 Appending the positive and the negative off-surface points along the surface normals at each point. 159

7.5 The isosurface extracted using the function learnt by Gaussian Process regression. The point cloud data is also shown. It took 6 minutes to learn this implicit surface form $10,000$ surface points. 160

8.1 (a) A full preference graph and (b) chain preference graph for a ranking problem with 4 classes. 167

8.2 Log-sigmoid lower bound for the 0-1 indicator function. 176

8.3 (a) Approximation of the sigmoid function $\sigma(z) \approx 1-\frac{1}{2}\text{erfc}(\frac{\sqrt{3}z}{\sqrt{2}\pi})$. (b) The erfc function. 177

8.4 *Effect of ϵ-exact derivatives* (a) The time taken and (b) the WMW statistic for the proposed method and the faster version of the proposed method as a function of ϵ. The CG tolerance was set to 10^{-3}. Results are for dataset 10. 185

Chapter 1

Introduction

During the past few decades it has become relatively easy to gather huge amounts of data, which are often apprehensively called *massive data sets.* According to a recent estimate in 2006 about 161 billion gigabytes of digital information was created. A few examples include datasets in genome sequencing, astronomical databases, internet databases, experimental data from particle physics, medical databases, financial records, weather reports, audio and video data. A goal in these areas is to build systems which can automatically extract useful information from the raw data. *Learning* is a principled method for distilling *predictive* and therefore scientific theories from the data [55].

1.1 Computational curse of non-parametric methods

The *parametric approach* to learning assumes a functional form for the model to be learnt, and then estimates the unknown parameters. Once the model has been trained *the training examples can be discarded.* The essence of the training examples have been captured in the model parameters, using which we can draw further inferences. However, unless the form of the model is known a priori, assuming it very often leads to erroneous inference.

Nonparametric methods do not make any assumptions on the form of the

underlying model. This is sometimes referred to as *'letting the data speak for themselves'* [87]. A price to be paid is that all the available *data has to be retained* while making the inference. It should be noted that nonparametric does not mean a lack of parameters, but rather that the underlying function/model of a learning problem cannot be indexed with a finite number of parameters. The number of parameters usually grows with the size of the training data. These are also known as *memory based methods*–the model is the entire training set.

One of the major bottlenecks for successful inference using nonparametric methods is their computational complexity. Most of the current state-of-the-art nonparametric machine learning algorithms have the computational complexity of either $\mathcal{O}(N^2)$ (for prediction at N points) or $\mathcal{O}(N^3)$ (for training), where N is the number of training examples. This has seriously restricted the use of massive data sets. For example, a simple kernel density estimation with 1 million points would take around 2 days.

1.2 Bringing computational tractability to massive datasets

Following are the two commonly used strategies on which much research has been done in order to cope with this quadratic scaling.

1. *Subset of data* These methods are based on using a small representative subset of the training examples. Different schemes specify different strategies to effectively choose the subset [91, 77, 19, 45, 44, 14, 82, 81, 79]. These methods can be considered to provide *exact inference in an approximate model.* While

these methods are often useful in practice they do not provide firm theoretical guarantees.

2. *Online learning* This strategy uses sequential update methods which can find good solutions in single passes through the data. This cuts down the need for running very large scale batch optimizers.

This thesis takes a different novel approach to this problem. At the heart of various algorithms is the multiplication of a structured matrix with a vector, which we refer to as *matrix-vector product* (MVP) primitive. This MVP is the bottleneck contributing to the $\mathcal{O}(N^2)$ quadratic complexity. I use ideas and techniques from computational physics (fast multipole methods), scientific computing (Krylov subspace methods), and computational geometry (*kd*-trees,clustering) to speed up *approximate* calculation of these primitives to $\mathcal{O}(N)$ and also provide *high accuracy guarantees*. In analogy these methods provide *approximate inference in an exact model*.

1.3 Weighted superposition of kernels

In most kernel based machine learning algorithms [71], Gaussian processes [58], and nonparametric statistics [39] the key computationally intensive task is to compute a linear combination of local kernel functions centered on the training data, *i.e.*,

$$f(x) = \sum_{i=1}^{N} q_i k(x, x_i), \tag{1.1}$$

where,

- $\{x_i \in \mathbb{R}^d, i = 1, \ldots, N\}$ are the N training data points,
- $\{q_i \in \mathbb{R}, i = 1, \ldots, N\}$ are the appropriately chosen weights,
- $k : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$ is the local kernel function,
- and $x \in \mathbb{R}^d$ is the test point at which f is to be computed.

The computational complexity to evaluate Equation 1.1 at a given test point is $\mathcal{O}(N)$.

For kernel machines (e.g. regularized least squares [55], support vector machines [13], kernel regression [87]) f is the regression/classification function. This is a consequence of the well known classical representer theorem [84] which states that the solutions of certain risk minimization problems involving an empirical risk term and a quadratic regularizer can be written as expansions in terms of the kernels centered on the training examples. In case of Gaussian process regression [90] f is the mean prediction. For non-parametric density estimation it is the kernel density estimate [87].

Training these models scales as $\mathcal{O}(N^3)$ since most involve solving a linear system of equations of the form

$$(\mathbf{K} + \lambda \mathbf{I})\xi = \mathbf{y}, \tag{1.2}$$

where, $\mathbf{K}$ is the $N \times N$ Gram matrix where $[\mathbf{K}]_{ij} = k(x_i, x_j)$, λ is some regularization parameter or noise variance, and $\mathbf{I}$ is the identity matrix. For specific kernel methods then there are many published techniques for speeding things up. However a naive implementation would scale as $\mathcal{O}(N^3)$.

Also many kernel methods in unsupervised learning like kernel principal component analysis [78], spectral clustering [11], and Laplacian eigenmaps involve computing the eigen values of the Gram matrix. Solutions to such problems can be obtained using iterative methods, which scales as $\mathcal{O}(N^2)$.

Most kernel methods also require choosing some parameters (e.g. bandwidth h of the kernel). Optimal procedures to choose these parameters cost $\mathcal{O}(N^2)$. The dominant computation there is also evaluation of $f(x)$ [61].

Recently, such nonparametric problems have been collectively referred to as *N-body problems in learning* by [25], in analogy with the coulombic, magnetostatic, and gravitational N-body potential problems arising in computational physics [27], where all pairwise interactions in a large ensemble of particles must be calculated.

1.4 Fast approximate matrix-vector product

In general we need to evaluate Equation 1.1 at M points $\{y_j \in \mathbb{R}^d, j = 1, \ldots, M\}$, *i.e.*,

$$f(y_j) = \sum_{i=1}^{N} q_i k(y_j, x_i) \;\; j = 1, \ldots, M, \tag{1.3}$$

leading to the quadratic $\mathcal{O}(MN)$ cost. We will develop fast ϵ-exact algorithms that compute the sum (1.3) approximately in linear $\mathcal{O}(M + N)$ time. The algorithm is ϵ-exact in the sense made precise below.

For any given $\epsilon > 0$, $\widehat{f}$ is an $\epsilon - exact$ approximation to f if the maximum absolute error relative to the total weight $Q = \sum_{i=1}^{N} |q_i|$ is upper bounded by ϵ, i.e.,

$$\max_{y_j} \left[\frac{|\hat{f}(y_j) - f(y_j)|}{Q} \right] \leq \epsilon. \tag{1.4}$$

The constant in $\mathcal{O}(M+N)$, depends on the desired *accuracy* ϵ, which however can be *arbitrary.* In fact for machine precision accuracy there is no difference between the results of the direct and the fast methods.

The sum in equation 1.3 can be thought of as a *matrix-vector multiplication* $f = \mathbf{K}q$, where $\mathbf{K}$ is a $M \times N$ matrix the entries of which are of the form $[\mathbf{K}]_{ij} = k(y_j, x_i)$ and $q = [q_1, \ldots, q_N]^T$ is a $N \times 1$ column vector.

A dense matrix of order $M \times N$ is called a *structured matrix* if its entries depend only on $\mathcal{O}(M+N)$ parameters. Philosophically, the reason we will be able to achieve $\mathcal{O}(M+N)$ algorithms to compute the matrix-vector multiplication is that the matrix $\mathbf{K}$ is a structured matrix, since all the entries of the matrix are determined by the set of $M+N$ points $\{x_i\}_{i=1}^N$ and $\{y_j\}_{i=1}^M$. If the entries of the of the matrix $\mathbf{K}$ were completely random than we could not do any better than $\mathcal{O}(MN)$.

1.5 Fast multipole methods

The fast algorithm is based on series expansion of the kernel and retaining only the first few terms contributing to the desired accuracy. The algorithms are in the spirit of *fast multipole methods* used in computational physics. The fast multipole method has been called one of the ten most significant algorithms [17] in scientific computation discovered in the 20th century, and won its inventors, Vladimir Rokhlin and Leslie Greengard, the 2001 Steele prize. Originally this method was developed for the fast summation of the potential fields generated by a large number of sources

(charges), such as those arising in gravitational or electrostatic potential problems, that are described by the Laplace equation in two or three dimensions [28]. The expression for the potential of a source located at a point can be factored in terms of an expansion containing the product of *multipole* functions and *regular* functions. This lead to the name for the algorithm. Since then FMM has also found application in many other problems, for example, in electromagnetic scattering, radial basis function fitting, molecular and stellar dynamics, and can be viewed as a fast matrix-vector product algorithm for particular structured matrices.

1.6 Motivating example–polynomial kernel

We will motivate the main idea using a simple polynomial kernel that is often used in kernel methods. The polynomial kernel of order p is given by

$$k(x, y) = (x \cdot y + c)^p. \tag{1.5}$$

Direct evaluation of the sum $f(y_j) = \sum_{i=1}^{N} q_i k(x_i, y_j)$ at M points requires $\mathcal{O}(MN)$ operations. The reason for this is that for each term in the sum the x_i and y_j appear together and hence we have to do all pair-wise operations. We will compute the same sum in $\mathcal{O}(M+N)$ time by *factorizing* the kernel and *regrouping* the terms. The polynomial kernel can be written as follows using the binomial theorem.

$$k(x, y) = (x \cdot y + c)^p = \sum_{k=0}^{p} \begin{pmatrix} p \\ k \end{pmatrix} (x \cdot y)^k c^{p-k}. \tag{1.6}$$

Also for simplicity let x and y be scalars, *i.e.*, $x, y \in \mathbb{R}$. As a result we have $(x \cdot y)^k = x^k y^k$. The multivariate case can be handled using multi-index notation

and will be discussed later. So now the sum – after suitable *regrouping* – can be written as follows:

$$f(y_j) = \sum_{i=1}^{N} q_i \left[\sum_{k=0}^{p} c^{p-k} \begin{pmatrix} p \\ k \end{pmatrix} x_i^k y_j^k \right] = \sum_{k=0}^{p} c^{p-k} \begin{pmatrix} p \\ k \end{pmatrix} y_j^k \left[\sum_{i=1}^{N} q_i x_i^k \right]$$
$$= \sum_{k=0}^{p} c^{p-k} \begin{pmatrix} p \\ k \end{pmatrix} y_j^k M_k, \qquad (1.7)$$

where $M_k = \sum_{i=1}^{N} q_i x_i^k$, can be called the *moments*. The moments $M_0, \ldots, M_p$ can be precomputed in $\mathcal{O}(pN)$ time. Hence f can be computed in linear $\mathcal{O}(pN + pM)$ time. This is sometimes known as encapsulating information in terms of the moments. Also note that for this simple kernel the sum was computed exactly.

In general any kernel $k(x, y)$ can be expanded in some region as

$$k(x, y) = \sum_{k=0}^{p} \Phi_k(x) \Psi_k(y) + error, \qquad (1.8)$$

where the function Φ_k depends only on x and Ψ_k on y. We call p, the *truncation number*–which has to be chosen such that the error is less than the desired accuracy ϵ. The fast summation – after suitable regrouping – is of the form

$$f(y_j) = \sum_{k=0}^{p} A_k \Psi_k(y) + error, \qquad (1.9)$$

where the moments A_k can be pre-computed as $A_k = \sum_{i=1}^{N} q_i \Phi_k(x_i)$. Using series expansions about a single point can lead to large truncation numbers. We need to organize the datapoints into different clusters using data-structures and use series expansion about the cluster centers. Also we need to give accuracy guarantees. So there are two aspects to this problem

1. Approximation theory → series expansions and error bounds.

2. Computational geometry → effective data-structures.

1.7 Thesis contributions

The thesis consists of two core contributions (1) design of fast summation algorithms and (2) applying these fast primitives to certain large scale machine learning problems. Table 1.1 summarizes the fast algorithms developed in this thesis and the tasks to which they were applied.

The rest of the thesis is organized as follows. In the next three chapters I describe three core algorithms for three different kernels–(1) the Gaussian, (2) Hermite times Gaussian, and (3) the error function. The applications are discussed in detail after the core algorithms have been explained.

The source code for all the fast summation algorithms are released under the GNU Lesser General Public License (LGPL). The source code can be downloaded from `http://www.umiacs.umd.edu/~vikas/Software/software.html`.

Kernel	Core MVP primitive	Applications
Gaussian	$G(y_j) = \sum_{i=1}^{N} q_i e^{-\|y_j - x_i\|^2/h^2}$	kernel density estimation
	Chapter 2	Chapter 5
		Gaussian process regression
		Chapter 7
		implicit surface fitting
		Chapter 7
Hermite×	$G(y_j) = \sum_{i=1}^{N} q_i H_r \left(\frac{y_j - x_i}{h_1} \right) e^{-(y_j - x_i)^2/h_2^2}$	optimal bandwidth estimation
Gaussian	Chapter 3	Chapter 6
		projection pursuit
		Chapter 6
error	$G(y_j) = \sum_{i=1}^{N} q_i \operatorname{erfc}(y_j - x_i)$	ranking
function	Chapter 4	Chapter 8
		collaborative filtering
		Chapter 8

Table 1.1: *Summary of the thesis.* The fast summation algorithms designed and tasks to which they were applied. Computation of each of these primitives at M points requires $\mathcal{O}(MN)$ time. The fast algorithms we design computes the same to a specified ϵ accuracy in $\mathcal{O}(M + N)$ time.

Chapter 2

Algorithm 1: Fast weighted summation of multivariate Gaussians

Evaluating sums of multivariate Gaussian kernels is a key computational task in many problems in computational statistics and machine learning. The computational cost of the direct evaluation of such sums scales as the product of the number of kernel functions and the evaluation points. The fast Gauss transform proposed by [29] is a ϵ-exact approximation algorithm that reduces the computational complexity of the evaluation of the sum of N Gaussians at M points in d dimensions from $\mathcal{O}(MN)$ to $\mathcal{O}(M+N)$. However, the constant factor in $\mathcal{O}(M+N)$ grows exponentially with increasing dimensionality d, which makes the algorithm impractical for dimensions greater than three. In this chapter we present a new algorithm where the constant factor is reduced to asymptotically polynomial order. The reduction is based on a new multivariate Taylor series expansion scheme combined with the efficient space subdivision using the k-center algorithm. We also integrate the algorithm with a kd-tree based nearest neighbor search. As a result the algorithm shows good performance both at small and large bandwidths. Our experimental results indicate that the proposed algorithm gives good speedups in dimensions as high as tens for moderate bandwidths and as high as hundreds for large and small bandwidths. [68, 64, 65, 59]

2.1 Discrete Gauss transform

The most commonly used kernel function is the *Gaussian kernel*

$$K(x,y) = e^{-\|x-y\|^2/h^2}, \tag{2.1}$$

where h is called the *bandwidth.* The bandwidth h controls the degree of smoothing, of noise tolerance, or of generalization.

The sum of multivariate Gaussian kernels is known as the *discrete Gauss transform* in the scientific computing literature. More formally, for each *target point* $\{y_j \in \mathbf{R}^d\}_{j=1,\ldots,M}$ the discrete Gauss transform is defined as,

$$G(y_j) = \sum_{i=1}^{N} q_i e^{-\|y_j - x_i\|^2/h^2}, \tag{2.2}$$

where $\{q_i \in \mathbf{R}\}_{i=1,\ldots,N}$ are the *source weights*, $\{x_i \in \mathbf{R}^d\}_{i=1,\ldots,N}$ are the *source points*, *i.e.*, the center of the Gaussians, and $h \in \mathbf{R}^+$ is the *source scale* or *bandwidth.* In other words $G(y_j)$ is the total contribution at y_j of N Gaussians centered at x_i each with bandwidth h. Each Gaussian is weighted by the term q_i.

The computational complexity to evaluate the discrete Gauss transform (Equation (2.2)) at M target points is $\mathcal{O}(MN)$. This makes the computation for large scale problems prohibitively expensive. In many machine learning tasks data-sets containing more than a million points are already common and larger problems are of interest.

The sum (2.2) can be thought of as a *matrix-vector multiplication* $\mathbf{K}q$, where $\mathbf{K}$ is a $M \times N$ matrix whose entries are of the form $[\mathbf{K}]_{ij} = k(y_j, x_i) = e^{-\|y_j - x_i\|^2/h^2}$ and $q = [q_1, \ldots, q_N]^T$ is a $N \times 1$ column vector. In this chapter we present a *fast*

algorithm that computes the matrix-vector multiplication *approximately* in linear $\mathcal{O}(M+N)$ time. The algorithm is approximate in the sense made precise below.

For any given $\epsilon > 0$, $\widehat{G}$ is an $\epsilon - exact$ approximation to G if the maximum absolute error relative to the total weight $Q = \sum_{i=1}^{N} |q_i|$ is upper bounded by ϵ, *i.e.*,

$$\max_{y_j} \left[\frac{|\hat{G}(y_j) - G(y_j)|}{Q} \right] \leq \epsilon. \tag{2.3}$$

The constant in $\mathcal{O}(M+N)$, depends on the desired *accuracy* ϵ, which however can be *arbitrary*. In fact for machine precision accuracy there is no difference between the results of the direct and the fast methods.

2.2 Related work

Before we present our algorithm we will briefly review the various approaches that have been proposed in the past to speedup the matrix-vector product. To simplify the exposition, in this section we assume $M = N$.

2.2.1 Methods based on sparse data-set representation

There are many strategies for specific problems which try to reduce this computational complexity by searching for a sparse representation of the data [91, 77, 19, 45, 44, 14, 82, 81, 79]. Most of these methods try to find a reduced subset of the original data-set using either random selection or greedy approximation. In most of these methods there is no guarantee on the approximation of the kernel matrix-vector product in a deterministic sense. These methods can be considered to provide *exact inference in an approximate model.*

2.2.2 Binned Approximation based on the FFT

If the source points are on an evenly spaced grid then we can compute the Gauss transform in $\mathcal{O}(N \log N)$ operations using the fast Fourier transform (FFT). One of the earliest methods, especially proposed for univariate fast kernel density estimation was based on this idea [74]. For irregularly spaced data, the space is divided into boxes, and the data is assigned to the closest neighboring grid points to obtain grid counts. The Gauss transform is also evaluated at regular grid points. For target points not lying on the the grid the value is obtained by interpolation based on the values at the neighboring grid points. The error introduced by interpolation reduces the accuracy of such methods. Also another problem with this method in higher dimensions is that the number of grid points grows exponentially with dimension.

2.2.3 Dual-tree methods

Dual-tree methods [25, 26] are based on space partitioning trees for both the source and target points. This method first builds a spatial tree like kd-trees or ball trees on both the source and target points. Using the tree data structure distance bounds between nodes can be computed. The bounds can be tightened by recursing on both trees. An advantage of the dual-tree methods is that they work for all common radial-basis kernel choices, not necessarily Gaussian. Dual-tree methods give good speed up only for small bandwidths. For moderate bandwidths they end up doing the same amount of work as the direct summation. These methods do

give accuracy guarantees. The single tree version takes $\mathcal{O}(N \log N)$ time while the dual-tree version is postulated to be $\mathcal{O}(N)$.

2.2.4 Fast Gauss transform

The *Fast Gauss Transform* (FGT) is an $\epsilon - exact$ approximation algorithm that reduces the computational complexity to $\mathcal{O}(N)$, at the expense of reduced precision, which however can be arbitrary. The constant depends on the desired precision, dimensionality of the problem, and the bandwidth. Given any $\epsilon > 0$, it computes an approximation $\hat{G}(y_j)$ to $G(y_j)$ such that the maximum absolute error relative to the total weight $Q = \sum_{i=1}^{N} |q_i|$ is upper bounded by ϵ.

The FGT was first proposed by [29] and applied successfully to a few lower dimensional applications in mathematics and physics. It uses a local representation of the Gaussian based on conventional Taylor series, a far field representation based on Hermite expansion, and translation formulae for conversion between the two representations. However the algorithm has not been widely used much in statistics, pattern recognition, and machine learning applications where higher dimensions occur commonly. An important reason for the lack of use of the algorithm in these areas is that the performance of the proposed FGT degrades exponentially with increasing dimensionality, which makes it impractical for these applications. The constant in the linear asymptotic cost $\mathcal{O}(N)$ grows roughly as p^d, *i.e.*, exponential in the dimension d. There are three reasons contributing to the degradation of the FGT in higher dimensions:

1. The number of the terms in the Hermite expansion used by the FGT grows exponentially with dimensionality, which causes the constant factor associated with the asymptotic complexity $\mathcal{O}(N)$ to increase exponentially with dimensionality.

2. The space subdivision scheme used by the fast Gauss transform is a uniform box subdivision scheme which is tolerable in lower dimensions but is extremely inefficient in higher dimensions.

3. The constant term due to the translation of the far-field Hermite series to the local Taylor series grows exponentially quickly with dimension making it impractical for dimensions greater than three.

2.3 Improved fast Gauss transform

In this chapter we present an *improved fast Gauss transform* (IFGT) suitable for higher dimensions. For the IFGT the constant term is asymptotically polynomial in d, *i.e*, is grows roughly as d^p (see Figure 2.1) . The IFGT differs from the FGT in the following three ways, addressing each of the issues above.

1. A single multivariate Taylor series like expansion is used to reduce the number of the expansion terms to polynomial order.

2. The k-center algorithm is applied to subdivide the space which is more efficient in higher dimensions.

3. Our expansion can act both as a far-field and local expansion. As a result we

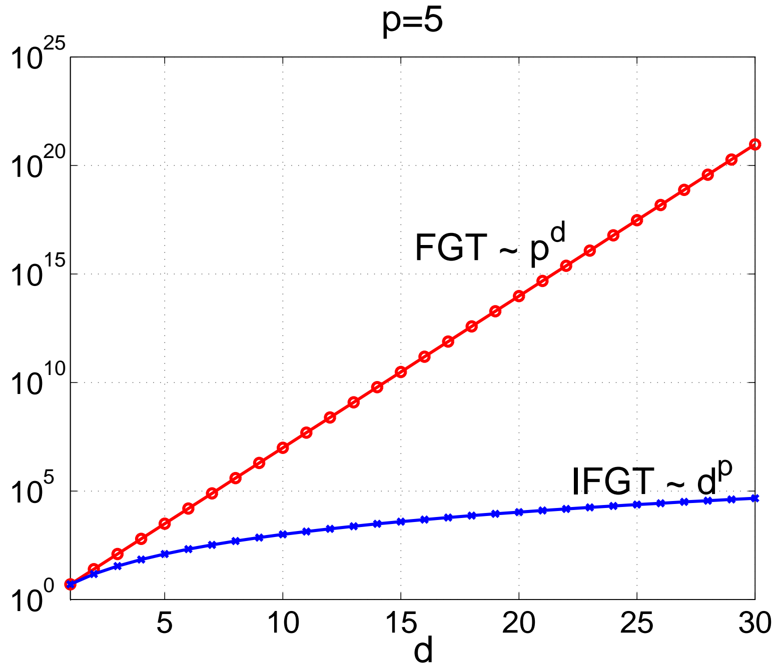


Figure 2.1: The constant term for the FGT and the IFGT complexity as a function of the dimensionality d. The FGT is exponential in d, while the IFGT is polynomial in d.

do not have separate far-field and local expansions which eliminates the cost of translation.

Our previous experiments reported that the IFGT did not give good speedups for small bandwidths. We integrate the IFGT algorithm with a kd-tree based nearest neighbor search. As a result we are able to obtain good speedups for both large and small bandwidths. We call the combined algorithm **FIGTree** – **F**ast **I**mproved **G**auss **T**ransform with kd-**Tree**.

The rest of the chapter is organized as follows. In Section 2.4 we introduce the key technical concepts used in the IFGT algorithm. More specifically, we discuss the multivariate Taylor series used to factorize the Gaussian, the multi-index notation,

and the space subdivision scheme based on the k-center clustering algorithm. In Section 2.5 we describe our improved fast Gauss transform and present runtime and storage analysis. In Section 2.6 we propose a strategy to choose the free parameters. In Section 2.7 we show how the IFGT can be integrated with a kd-tree based nearest neighbor search algorithm. In Section 8.4.2 we elucidate in detail how our current method differs from the fast Gauss transform. In Section 8.6 we present numerical results of our algorithm and compare it with the dual-tree algorithms.

2.4 Preliminaries

Before we discuss the IFGT we first discuss the multivariate Taylor series expansion, multi-index notation, and our space subdivision scheme.

2.4.1 Multidimensional Taylor Series

The factorization of the multivariate Gaussian and the evaluation of the error bounds are based on the multidimensional Taylor series and Lagrange evaluation of the remainder which we state here without the proof.

Theorem 1 *[Taylor series] For any point $x_* \in \mathbf{R}^d$, let $I \subset \mathbf{R}^d$ be an open set containing the point x_*. Let $f : I \to \mathbf{R}$ be a real valued function which is n times partially differentiable on I. Then for any $x = (x_1, x_2, \ldots, x_d) \in I$, there is a $\theta \in \mathbf{R}$ with $0 < \theta < 1$ such that*

$$f(x) = \sum_{k=0}^{n-1} \frac{1}{k!} \left[(x - x_*) \cdot \nabla\right]^k f(x_*) + \frac{1}{n!} \left[(x - x_*) \cdot \nabla\right]^n f(x_* + \theta(x - x_*)), \quad (2.4)$$

where $\nabla = \left(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \ldots, \frac{\partial}{\partial x_d}\right)$ *and the operator* $[(x - x_*) \cdot \nabla]^k$ *operates as*

$$[(x-x_*)\cdot\nabla]^k = \sum_{\substack{0 \le i_1,\ldots,i_d \le k \\ i_1+\ldots+i_d=k}} \frac{k!}{i_1!\cdots i_d!}(x_1-x_{*1})^{i_1}\cdots(x_d-x_{*d})^{i_d}\frac{\partial^k}{\partial^{i_1}\ldots\partial^{i_d}}. \tag{2.5}$$

Based on the above theorem we have the following theorem which gives the multivariate Taylor series expansion of the exponential function $e^{2(x-x_*).(y-x_*)/h^2}$.

Theorem 2 *Let* $B_{r_x}(x_*)$ *be a open ball of radius* r_x *with center* $x_* \in \mathbf{R}^d$*, i.e.,* $B_{r_x}(x_*) = \{x \in \mathbf{R}^d : \|x - x_*\| < r_x\}$*. Let* $h \in \mathbf{R}^+$ *be a positive constant and* $y \in \mathbf{R}^d$ *be a fixed point such that* $\|y - x_*\| < r_y$*. For any* $x \in B_{r_x}(x_*)$ *and any non-negative integer* p *the function* $f(x) = e^{2(x-x_*).(y-x_*)/h^2}$ *can be written as*

$$f(x) = e^{2(x-x_*).(y-x_*)/h^2} = \sum_{k=0}^{p-1}\frac{2^k}{k!}\left[\left(\frac{x-x_*}{h}\right)\cdot\left(\frac{y-x_*}{h}\right)\right]^k + R_p(x), \tag{2.6}$$

and the residual $R_p(x)$ *is bounded as follows.*

$$|R_p(x)| \le \frac{2^p}{p!}\left(\frac{\|x-x_*\|}{h}\right)^p\left(\frac{\|y-x_*\|}{h}\right)^p e^{2\|x-x_*\|\|y-x_*\|/h^2}. \tag{2.7}$$

$$< \frac{2^p}{p!}\left(\frac{r_x r_y}{h^2}\right)^p e^{2r_x r_y/h^2}. \tag{2.8}$$

Proof : Let us define a new function $g(x) = e^{2[x.(y-x_*)]/h^2}$. Using the result

$$[(x-x_*)\cdot\nabla]^k g(x_*) = 2^k e^{2[x_*.(y-x_*)]/h^2}\left[\left(\frac{x-x_*}{h}\right)\cdot\left(\frac{y-x_*}{h}\right)\right]^k \tag{2.9}$$

and Theorem 4, we have for any $x \in B_{r_x}(x_*)$ there is a $\theta \in \mathbf{R}$ with $0 < \theta < 1$ such that

$$\begin{aligned} g(x) &= e^{2[x_*.(y-x_*)]/h^2} \left\{ \sum_{k=0}^{p-1} \frac{2^k}{k!} \left[\left(\frac{x-x_*}{h} \right) \cdot \left(\frac{y-x_*}{h} \right) \right]^k \right. \\ &\left. + \frac{2^p}{p!} \left[\left(\frac{x-x_*}{h} \right) \cdot \left(\frac{y-x_*}{h} \right) \right]^p e^{2\theta[(x-x_*).(y-x_*)]/h^2} \right\}. \end{aligned}$$

Hence

$$f(x) = e^{2(x-x_*).(y-x_*)/h^2} = \sum_{k=0}^{p-1} \frac{2^k}{k!} \left[\left(\frac{x-x_*}{h} \right) \cdot \left(\frac{y-x_*}{h} \right) \right]^k + R_p(x), \qquad (2.10)$$

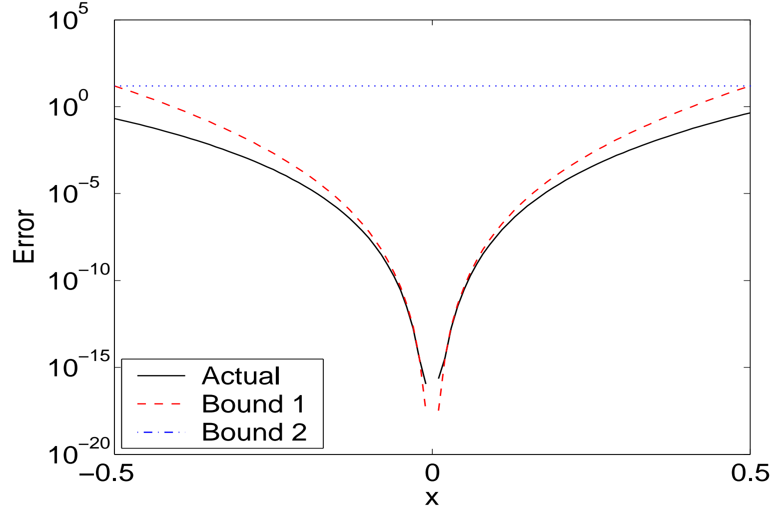
where,

$$R_p(x) = \frac{2^p}{p!} \left[\left(\frac{x-x_*}{h} \right) \cdot \left(\frac{y-x_*}{h} \right) \right]^p e^{2\theta[(x-x_*).(y-x_*)]/h^2}. \qquad (2.11)$$

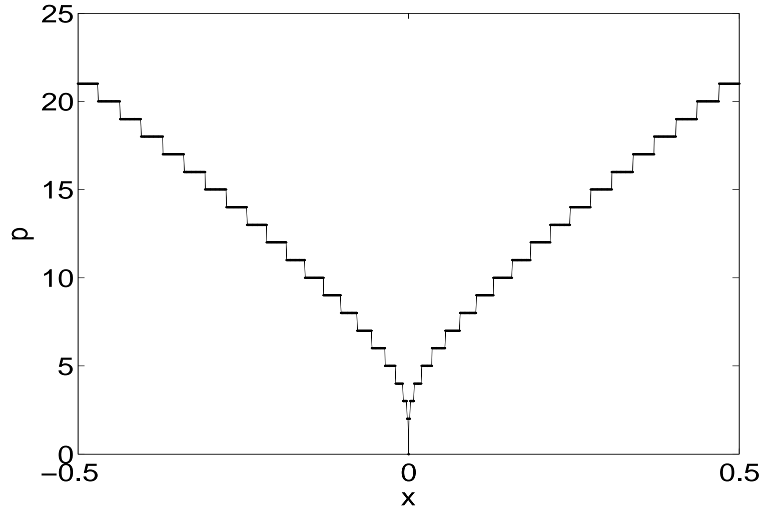
Using the Cauchy-Schwartz inequality $x \cdot y \leq \|x\|\|y\|$ the remainder is bounded as follows.

$$\begin{aligned} |R_p(x)| &= \frac{2^p}{p!} \left[\left(\frac{x-x_*}{h} \right) \cdot \left(\frac{y-x_*}{h} \right) \right]^p e^{2\theta[(x-x_*).(y-x_*)]/h^2}, \\ &\leq \frac{2^p}{p!} \left(\frac{\|x-x_*\|}{h} \right)^p \left(\frac{\|y-x_*\|}{h} \right)^p e^{2\theta\|x-x_*\|\|y-x_*\|/h^2}, \\ &\leq \frac{2^p}{p!} \left(\frac{\|x-x_*\|}{h} \right)^p \left(\frac{\|y-x_*\|}{h} \right)^p e^{2\|x-x_*\|\|y-x_*\|/h^2} [\text{Since } \ 0 < \theta < 1], \\ &< \frac{2^p}{p!} \left(\frac{r_x r_y}{h^2} \right)^p e^{2r_x r_y/h^2} \quad [\text{Since } \|x-x_*\| < r_x \text{ and } \|y-x_*\| < r_y]. \qquad (2.12) \end{aligned}$$

Remark: Figure 2.2(a) compares the actual residual and the bound given by (2.7) as a function of x, for $p = 10$ and $d = 1$. The actual residual $R_p(x)$ vanishes at $x = x_*$ and increases as x moves away from x_*. The dashed line shows the bound given by (2.7). It can be seen that the bound is quite tight in practice. The dotted line is the bound which is independent of x (Equation (2.12)). It can be seen that this bound is very pessimistic for $\|x - x_*\| < r_x$. A consequence of the use of (2.7)



(a)



(b)

Figure 2.2: (a) The actual residual (solid line) and the bound (dashed line) given by Equation (2.7) as a function of x. [$x_* = 0$, $y = 1.0$, $h = 0.5$, $r_x = 0.5$, $r_y = 1.0$, and $p = 10$]. The residual is minimum at $x = x_*$ and increases as x moves away from x_*. The dotted line shows the very pessimistic bound which is independent of x (Equation (2.12)) used in the original IFGT. (b) The truncation number p required as a function of x so that the error is less than10^{-6}.

is that a lower truncation number p can achieve a given error, for smaller $\|x - x_*\|$. Figure 2.2(b) shows the truncation number p required as the function of x so that the error is less than 10^{-6}. It can be seen that for points close to x_* we need a very small truncation number compared to points far from the center. The original IFGT and the FGT algorithms used the same truncation number for all the points in the open ball. The truncation number was thus large as it was chosen based on the points at the boundary. However our current approach adaptively chooses p based on the actual values of $\|x - x_*\|$.

2.4.2 Multi-index Notation

In order to manipulate the multivariate terms in the Taylor series we will need the notion of multi-indices.

- A multi-index $\alpha = (\alpha_1, \alpha_2 \ldots, \alpha_d) \in \mathbb{N}^d$ is a d-tuple of nonnegative integers.
- The length of the multi-index α is defined as $|\alpha| = \alpha_1 + \alpha_2 + \ldots + \alpha_d$.
- The factorial of α is defined as $\alpha! = \alpha_1!\alpha_2! \ldots \alpha_d!$.
- For any multi-index $\alpha \in \mathbb{N}^d$ and $x = (x_1, x_2, \ldots, x_d) \in \mathbb{R}^d$ the *d-variate monomial* x^α is defined as $x^\alpha = x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_d^{\alpha_d}$.
- x^α is of *degree* n if $|\alpha| = n$.
- The total number of d-variate monomials of degree n is $\binom{n+d-1}{d-1}$.

- The total number of d-variate monomials of degree *less than or equal to* n is

$$r_{nd} = \sum_{k=0}^{n} \binom{k+d-1}{d-1} = \binom{n+d}{d}. \tag{2.13}$$

- Let $x, y \in \mathbb{R}^d$ and $v = x \cdot y = x_1 y_1 + \ldots + x_d y_d$. Then using the multi-index notation v^n can be written as,

$$v^n = (x \cdot y)^n = \sum_{|\alpha|=n} \frac{n!}{\alpha!} x^\alpha y^\alpha. \tag{2.14}$$

2.4.3 Space sub-division

In the IFGT we will appropriately cluster source points and evaluate their contributions using an expression that involves the Taylor series. Accordingly we need a strategy to choose a set of centers about which to expand the Taylor series, *i.e.*, we need to subdivide the space. We model the space subdivision task as a k-center problem, which is defined as follows:

k-center problem: *Given a set of N points in d dimensions and a predefined number of the clusters k, find a partition of the points into clusters $S_1, \ldots, S_k$, and also the cluster centers $c_1, \ldots, c_k$, so as to minimize the cost function-the maximum radius of clusters,* $\max_i \max_{x \in S_i} \|x - c_i\|$.

The k-center problem is known to be NP-hard [5]. Gonzalez [24] proposed a very simple greedy algorithm, called *farthest-point clustering*, and proved that it gives an approximation factor of 2.

Initially pick an arbitrary point v_0 as the center of the first cluster and add it to the center set C. Then for $i = 1$ to k do the following: at step i, for every point v, compute its distance to the set C: $d_i(v, C) = \min_{c \in C} \|v - c\|$. Let v_i be the point

that is farthest from C, i.e., the point for which $d_i(v_i, C) = \max_v d_i(v, C)$. Add v_i to set C. Report the points $v_0, v_1, \ldots, v_{k-1}$ as the cluster centers. Each point is assigned to its nearest center.

Gonzalez proved the following 2-approximation theorem for the farthest-point clustering algorithm [24] [1].

Theorem 3 *[24] For k-center clustering, the farthest-point clustering computes a partition with maximum radius at most twice the optimum.*

Proof: For completeness, we provide a simple proof for the above theorem. First note that the radius of the farthest-point clustering solution by definition is

$$d_k(v_k, C) = \max_v \min_{c \in C} \|v - c\|.$$

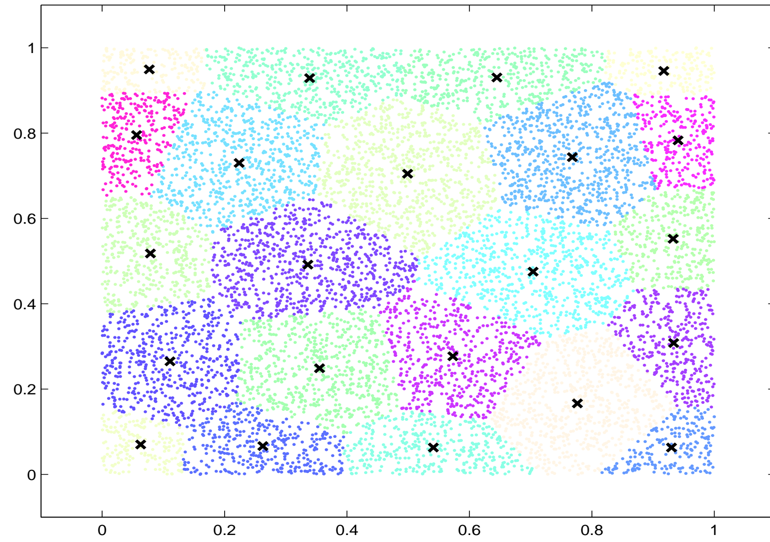
In the optimal k-center case, two of these $k + 1$ points, say v_i and v_j, must be in a same cluster centered at c by the pigeon hole principle. Observe that the distance from each point to the set C does not increase as the algorithm progresses. Therefore $d_k(v_k, C) \leq d_i(v_k, C)$ and $d_k(v_k, C) \leq d_j(v_k, C)$. Also by definition, we have $d_i(v_k, C) \leq d_i(v_i, C)$ and $d_j(v_k, C) \leq d_j(v_j, C)$. So we have

$$\|v_i - c\| + \|v_j - c\| \geq \|v_i - v_j\| \geq d_k(v_k, C),$$

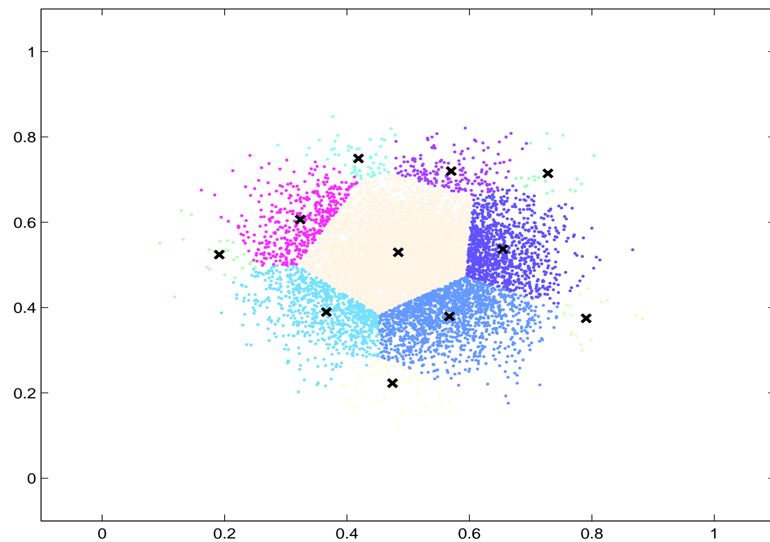
by the triangle inequality. Since $\|v_i - c\|$ and $\|v_j - c\|$ are both at most the optimal radius δ, we have the radius of the farthest-point clustering solution $d_k(v_k, C) \leq 2\delta$.

The direct implementation of farthest-point clustering has running time $\mathcal{O}(Nk)$. [18] gave a two-phase algorithm with optimal running time $\mathcal{O}(N \log k)$. The first

[1] It was proved in [37] that the factor 2 cannot be improved unless $P = NP$.



(a)



(b)

Figure 2.3: (a) Using the farthest-point clustering algorithm 10,000 points uniformly distributed in a unit square are divided into 22 clusters with the maximum radius of the clusters being 0.2. (b) 10,000 points normally distributed in a unit square are divided into 11 clusters with the maximum radius of the clusters being 0.2.

phase of their algorithm clusters points into rectangular boxes using Vaidya's [83] *box decomposition*– a sort of quadtree in which cubes are shrunk to bounding boxes before splitting. The second phase resembles the farthest-point clustering on a sparse graph that has a vertex for each box. In practice, the initial point has little influence on the final approximation radius, if number of the points is sufficiently large.

Figure 2.3 displays the results of farthest-point algorithm on sample two dimensional datasets. After the end of the clustering procedure the center of each cluster is recomputed as the mean of all the points lying in each cluster.

Remark: The farthest-point clustering algorithm is progressive. This means that if we have k centers and we wish to compute the $(k+1)^{th}$ center, the first k centers do not change. This is different from other clustering algorithms like the k-means. Our goal is not to get a very good clustering, but to organize the source points into spherical balls.

2.5 Improved Fast Gauss Transform

Having established the Taylor series expansion and the farthest-point clustering algorithm for k-center clustering, we are now ready to present the IFGT. The method relies on the expansion of the Gaussian using the truncated Taylor series

expansion. For any point $x_* \in \mathbf{R}^d$ the Gauss Transform at y_j can be written as,

$$
\begin{aligned}
G(y_j) &= \sum_{i=1}^{N} q_i e^{-\|y_j - x_i\|^2/h^2}, \\
&= \sum_{i=1}^{N} q_i e^{-\|(y_j - x_*) - (x_i - x_*)\|^2/h^2}, \\
&= \sum_{i=1}^{N} q_i e^{-\|x_i - x_*\|^2/h^2} e^{-\|y_j - x_*\|^2/h^2} e^{2(y_j - x_*)\cdot(x_i - x_*)/h^2}. \qquad (2.15)
\end{aligned}
$$

In Equation (3.3) the first exponential inside the summation $e^{-\|x_i - x_*\|^2/h^2}$ depends only on the source coordinates x_i. The second exponential $e^{-\|y_j - x_*\|^2/h^2}$ depends only on the target coordinates y_j. However in the third exponential $e^{2(y_j - x_*)\cdot(x_i - x_*)/h^2}$ the source and target are entangled. The crux of the algorithm is to separate this entanglement via the Taylor series expansion of this term.

2.5.1 Factorization

Using Theorem 5 the series expansion for $e^{2(y_j - x_*)\cdot(x_i - x_*)/h^2}$ can be written as,

$$
e^{2(y_j - x_*)\cdot(x_i - x_*)/h^2} = \sum_{n=0}^{p_i - 1} \frac{2^n}{n!} \left[\left(\frac{y_j - x_*}{h} \right) \cdot \left(\frac{x_i - x_*}{h} \right) \right]^n + error_{p_i}. \qquad (2.16)
$$

The truncation number p_i for each source x_i is chosen based on the prescribed error and the distance from the expansion center. A strategy for choosing p_i is discussed later. Using the multi-index notation (Equation (2.14)), this expansion can be written as,

$$
e^{2(y_j - x_*)\cdot(x_i - x_*)/h^2} = \sum_{|\alpha| \le p_i - 1} \frac{2^\alpha}{\alpha!} \left(\frac{y_j - x_*}{h} \right)^\alpha \left(\frac{x_i - x_*}{h} \right)^\alpha + error_{p_i}. \qquad (2.17)
$$

Ignoring error terms for now $G(y_j)$ can be approximated as,

$$\hat{G}(y_j) = \sum_{i=1}^{N} q_i e^{-\|x_i - x_*\|^2/h^2} e^{-\|y_j - x_*\|^2/h^2} \left[\sum_{|\alpha| \leq p_i - 1} \frac{2^\alpha}{\alpha!} \left(\frac{y_j - x_*}{h} \right)^\alpha \left(\frac{x_i - x_*}{h} \right)^\alpha \right]. \tag{2.18}$$

Let $p_{max} = \max_i p_i$ and $\mathbf{1}_{|\alpha| \leq p_i - 1}$ be an indicator function for $|\alpha| \leq p_i - 1$, that is,

$$\mathbf{1}_{|\alpha| \leq p_i - 1} = \begin{cases} 1 & \text{if } |\alpha| \leq p_i - 1 \\ 0 & \text{if } |\alpha| > p_i - 1 \end{cases}. \tag{2.19}$$

2.5.2 Regrouping

Rearranging the terms (2.18) can be written as

$$\begin{aligned} \hat{G}(y_j) &= \sum_{|\alpha| \leq p_{max} - 1} \left[\frac{2^\alpha}{\alpha!} \sum_{i=1}^{N} q_i e^{-\|x_i - x_*\|^2/h^2} \left(\frac{x_i - x_*}{h} \right)^\alpha \mathbf{1}_{|\alpha| \leq p_i - 1} \right] \\ &\quad e^{-\|y_j - x_*\|^2/h^2} \left(\frac{y_j - x_*}{h} \right)^\alpha, \\ &= \sum_{|\alpha| \leq p_{max} - 1} C_\alpha e^{-\|y_j - x_*\|^2/h^2} \left(\frac{y_j - x_*}{h} \right)^\alpha, \end{aligned} \tag{2.20}$$

where,

$$C_\alpha = \frac{2^\alpha}{\alpha!} \sum_{i=1}^{N} q_i e^{-\|x_i - x_*\|^2/h^2} \left(\frac{x_i - x_*}{h} \right)^\alpha \mathbf{1}_{|\alpha| \leq p_i - 1}. \tag{2.21}$$

The coefficients C_α can be evaluated separately in $\mathcal{O}(N)$. Evaluation of $\widehat{G}(y_j)$ at M points is $\mathcal{O}(M)$. Hence the computational complexity has reduced from the quadratic $\mathcal{O}(NM)$ to the linear $\mathcal{O}(N+M)$. A detailed analysis of the computational complexity is provided later.

2.5.3 Space subdivision

Thus far, we have used the Taylor series expansion about a certain point x_*. However if we use the same x_* for all the points we typically would require very

high truncation numbers since the Taylor series is valid only in a small open ball around x_*. We use an data adaptive space partitioning scheme like the farthest point clustering algorithm to divide the N sources into K clusters, S_k for $k = 1, \ldots, K$ with c_k being the center of each cluster. The Gauss transform can be written as,

$$\hat{G}(y_j) = \sum_{k=1}^{K} \sum_{|\alpha| \leq p_{max}-1} C_\alpha^k e^{-\|y_j - c_k\|^2/h^2} \left(\frac{y_j - c_k}{h} \right)^\alpha, \tag{2.22}$$

where,

$$C_\alpha^k = \frac{2^\alpha}{\alpha!} \sum_{x_i \in S_k} q_i e^{-\|x_i - c_k\|^2/h^2} \left(\frac{x_i - c_k}{h} \right)^\alpha \mathbf{1}_{|\alpha| \leq p_i - 1}. \tag{2.23}$$

2.5.4 Rapid decay of the Gaussian

Since the Gaussian decays very rapidly a further speedup is achieved if we ignore all the sources belonging to a cluster if the cluster is greater than a certain distance from the target point, $\|y_j - c_k\| > r_y^k$. The cluster cutoff radius depends on the desired precision ϵ. So now the Gauss transform is evaluated as

$$\hat{G}(y_j) = \sum_{\|y_j - c_k\| \leq r_y^k} \sum_{|\alpha| \leq p_{max}-1} C_\alpha^k e^{-\|y_j - c_k\|^2/h^2} \left(\frac{y_j - c_k}{h} \right)^\alpha, \tag{2.24}$$

where,

$$C_\alpha^k = \frac{2^\alpha}{\alpha!} \sum_{x_i \in S_k} q_i e^{-\|x_i - c_k\|^2/h^2} \left(\frac{x_i - c_k}{h} \right)^\alpha \mathbf{1}_{|\alpha| \leq p_i - 1}. \tag{2.25}$$

2.5.5 Runtime analysis

- The farthest point clustering algorithm has run time of $\mathcal{O}(dN \log K)$ [18].
- Since each source point belongs to only one cluster, computing the cluster coefficients C_α^k for all the clusters is of $\mathcal{O}(N r_{(p_{max}-1)d})$, where $r_{(p_{max}-1)d} =$

$\binom{p_{max}+d-1}{d}$ is the maximum number of coefficients for each cluster. It is the total number of d-variate monomials of degree less than or equal to $p_{max} - 1$.

- Computing $\hat{G}(y_j)$ is of $\mathcal{O}(Mnr_{(p_{max}-1)d})$ where n if the maximum number of neighbor clusters (depends on the bandwidth h and the error ϵ) which influence the target.

- We also need to account the cost needed to determine n. This involves looping through all K clusters and computing the distance between each of the M test points and each of the K cluster centers, resulting in an additional $\mathcal{O}(dKM)$ term. This term can be reduced if we use some efficient nearest neighbor search techniques [2].

Hence the total run time is

$$\mathcal{O}(dN \log K + Nr_{(p_{max}-1)d} + Mnr_{(p_{max}-1)d} + dKM). \tag{2.26}$$

Assuming $M = N$, the complexity is $\mathcal{O}(cN)$ – where the constant term

$$c = dK + d \log K + (1 + n)r_{(p_{max}-1)d} \tag{2.27}$$

depends on the *dimensionality*, the *bandwidth*, and the *accuracy* required. The number of terms $r_{(p_{max}-1)d}$ is asymptotically polynomial in d. For $d \to \infty$ and moderate p, the number of terms is approximately d^p.

A different truncation number is chosen for each data point depending on its distance from the cluster center. A good consequence of this strategy is that

[2] We present the FIGTree algorithm in Section 2.7 that reduces the cost of this search to $\mathcal{O}(d \log KM)$ using the ANN library [51].

only a few points at the boundary of the clusters have high truncation numbers. Theoretically we expect to get a much better speed up since for many points $p_i < p_{max}$. However some computation resources are used in determining the truncation numbers. As a result experimentally we saw a smaller improvement in speedup, compared to using the same truncation number for all points. The speedup is more noticeable in higher dimensions and for larger bandwidths.

2.5.6 Storage analysis

For each cluster we need to store $r_{(p_{max}-1)d}$ coefficients. Including the space required to store the source points, target points, and cluster centers the space required is

$$\mathcal{O}(Kr_{(p_{max}-1)d} + dN + dM + dK). \tag{2.28}$$

2.5.7 Efficient computation of multivariate polynomials–Horner's rule

Evaluating each d-variate monomial of degree n directly requires n multiplications. Hence direct evaluation of all d-variate monomials of degree less than or equal to n requires $\sum_{k=0}^{n} k\binom{k+d-1}{d-1}$ multiplications. The storage requirement is r_{nd}. However, efficient evaluation using the Horner's rule requires $r_{nd}-1$ multiplications [95]. The required storage is r_{nd} (See Table 2.1).

For a d-variate polynomial of order n, we can store all terms in a vector of length r_{nd}. Starting from the order zero term (constant 1), we take the following approach. Assume we have already evaluated terms of order $k-1$. We use an array

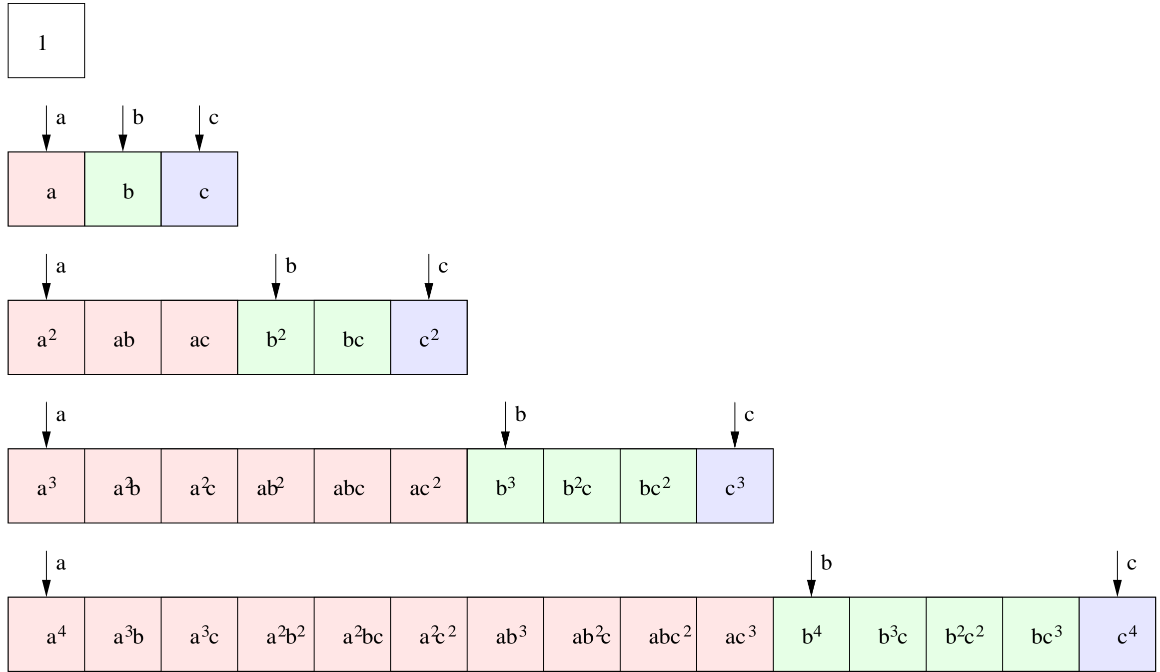


Figure 2.4: Efficient expansion of multivariate polynomials.

of size d to record the positions of the d leading terms (the simple terms such as a^{k-1}, b^{k-1}, c^{k-1}, . . . in Figure 2.4) in the terms of order $k-1$. Then terms of order k can be obtained by multiplying each of the d variables with all the terms between the variables leading term and the end, as shown in the Figure 2.4. The positions of the d leading terms are updated respectively. The required storage is r_{nd} and the computations of the terms require $r_{nd} - 1$ multiplications.

2.5.8 Partial distance

For each cluster we need to find the clusters which are within a certain radius to it. Computing partial distances helps to reduce the computational burden in nearest-neighbor searches in high dimensional spaces. By *partial distance*, we calculate the distance using some subset r of the coordinates from the full d dimensions. If this

n	2	4	6	8	10	12	15	20
Direct d=2	8	40	112	240	440	728	1360	3080
Efficient d=2	5	14	27	44	65	90	135	230
Direct d=3	15	105	378	990	2145	4095	9180	26565
Efficient d=3	9	34	83	164	285	454	815	1770
Direct d=6	48	720	4752	20592	68640	190944	697680	3946800
Efficient d=6	27	209	923	3002	8007	18563	54263	230229
Direct d=10	120	3640	43680	318240	1679600	7054320	44574000	546273000
Efficient d=10	65	1000	8007	43757	184755	646645	3268759	30045014

Table 2.1: Number of multiplications required for the direct and the efficient method for evaluating all d-variate monomials of degree less than or equal to n.

partial distance is too great we do not compute distances any further. The partial distance is strictly nondecreasing as we add the contributions from more and more dimensions.

2.6 Choosing the parameters

Given any $\epsilon > 0$, we want to choose the following parameters,

- K (the number of clusters),
- $\{r_y^k\}_{k=1}^K$ (the cut off radius for each cluster),
- and $\{p_i\}_{i=1}^N$ (the truncation number for each source point x_i)

such that for any target point y_j we can guarantee that

$$\frac{|\hat{G}(y_j) - G(y_j)|}{Q} \leq \epsilon, \tag{2.29}$$

where $Q = \sum_{i=1}^{N} |q_i|$.

Let us define Δ_{ij} to be the point wise error in $\widehat{G}(y_j)$ contributed by the ith source x_i. We now require that

$$|\hat{G}(y_j) - G(y_j)| = \left| \sum_{i=1}^{N} \Delta_{ij} \right| \leq \sum_{i=1}^{N} |\Delta_{ij}| \leq Q\epsilon = \sum_{i=1}^{N} |q_i|\epsilon. \quad (2.30)$$

One way to achieve this is to let

$$|\Delta_{ij}| \leq |q_i|\epsilon \ \ \forall i = 1, \ldots, N. \quad (2.31)$$

Let c_k be the center of the cluster to which x_i belongs. There are two different ways in which a source can contribute to the error.

- The first is due to ignoring the cluster S_k if it is outside a given radius r_y^k from the target point y_j. In this case,

$$\Delta_{ij} = q_i e^{-\|y_j - x_i\|^2/h^2} \ \text{ if } \|y_j - c_k\| > r_y^k. \quad (2.32)$$

- The second source of error is due to truncation of the Taylor series. For all clusters which are within a distance r_y^k from the target point the error is due to the truncation of the Taylor series after order p_i. From Equations 3.3 and 2.16 we have,

$$\Delta_{ij} = q_i e^{-\|x_i - c_k\|^2/h^2} e^{-\|y_j - c_k\|^2/h^2} error_{p_i} \ \text{ if } \|y_j - c_k\| \leq r_y^k. \quad (2.33)$$

Our strategy for choosing the parameters is as follows. The cutoff radius r_y^k for each cluster is chosen based on Equation (2.32) and the radius of each cluster r_x^k. Given r_y^k and $\|x_i - c_k\|$ the truncation number p_i for each source is chosen based on Equation (2.33). Towards the end we suggest a strategy to choose the number of clusters K and the maximum truncation p_{max} jointly.

2.6.1 Automatically choosing the cut off radius for each cluster

We ignore all sources belonging to a cluster S_k if $\|y_j - c_k\| > r_y^k$. r_y^k should be chosen such that for all sources in cluster S_k the error

$$|\Delta_{ij}| = |q_i| e^{-\|y_j - x_i\|^2/h^2} \leq |q_i|\epsilon. \tag{2.34}$$

This implies that

$$\|y_j - x_i\| > h\sqrt{\ln(1/\epsilon)} \tag{2.35}$$

Using the reverse triangle inequality, $\|a - b\| \geq \big|\|a\| - \|b\|\big|$, and the fact that $\|y_j - c_k\| > r_y^k$ and $\|x_i - c_k\| \leq r_x^k$, we have

$$\begin{aligned} \|y_j - x_i\| = \|y_j - c_k + c_k - x_i\| &= \|(y_j - c_k) - (x_i - c_k)\|, \\ &\geq \big|\|(y_j - c_k)\| - \|(x_i - c_k)\|\big|, \\ &> \big|r_y^k - r_x^k\big|. \end{aligned}$$

So in order that the error due to ignoring the faraway clusters is less than $q_i\epsilon$ we have to choose r_y^k and r_x^k such that,

$$\big|r_y^k - r_x^k\big| > h\sqrt{\ln(1/\epsilon)}. \tag{2.36}$$

If we choose $r_y^k > r_x^k$ then,

$$r_y^k > r_x^k + h\sqrt{\ln(1/\epsilon)}. \tag{2.37}$$

Let R be the maximum distance between any source and target point. For example if the data were distributed in a d-dimensional hypercube of length a, then $R \leq \sqrt{d}a$, i.e., the length of the maximum diagonal. Hence,

$$r_y^k > r_x^k + \min\left(R, h\sqrt{\ln(1/\epsilon)}\right). \tag{2.38}$$

2.6.2 Automatically choosing the truncation number for each source

From Theorem 5 we have,

$$error_{p_i} \leq \frac{2^{p_i}}{p_i!} \left(\frac{\|x_i - c_k\|}{h} \right)^{p_i} \left(\frac{\|y_j - c_k\|}{h} \right)^{p_i} e^{2\|x_i - c_k\| \|y_j - c_k\|/h^2}. \tag{2.39}$$

Hence for all sources for which $\|y_j - c_k\| \leq r_y^k$, substituting in Equation (2.33) we have

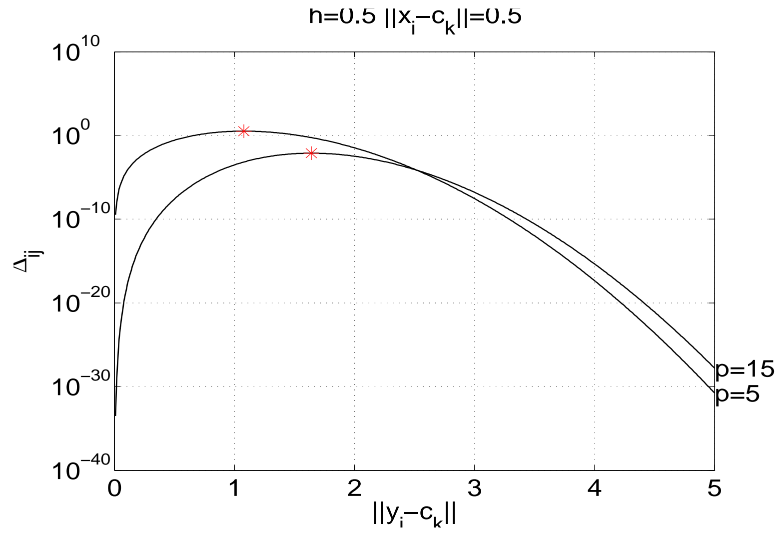
$$\Delta_{ij} \leq q_i \frac{2^{p_i}}{p_i!} \left(\frac{\|x_i - c_k\|}{h} \right)^{p_i} \left(\frac{\|y_j - c_k\|}{h} \right)^{p_i} e^{-(\|x_i - c_k\| - \|y_j - c_k\|)^2/h^2}. \tag{2.40}$$

For a given source x_i we have to choose p_i such that $|\Delta_{ij}| \leq |q_i|\epsilon$. Δ_{ij} depends both on distance between the source and the cluster center, i.e., $\|x_i - c_k\|$ and the distance between the target and the cluster center, i.e., $\|y_j - c_k\|$. The speedup is achieved because at each cluster S_k we sum up the effect of all the sources. As a result we do not have a knowledge of $\|y_j - c_k\|$ when we are using Equation (2.25). So we will have to bound the right hand side of Equation (2.40), such that it is independent of $\|y_j - c_k\|$. Figure 3.1 shows the error at y_j due to source x_i, i.e., Δ_{ij} [Equation (2.40)] as a function of $\|y_j - c_k\|$ for different values of p and for (a) $h = 0.5$ and (b) $h = 1.0$. The error increases as a function of $\|y_j - c_k\|$, reaches a maximum and then starts decreasing. The maximum is attained at

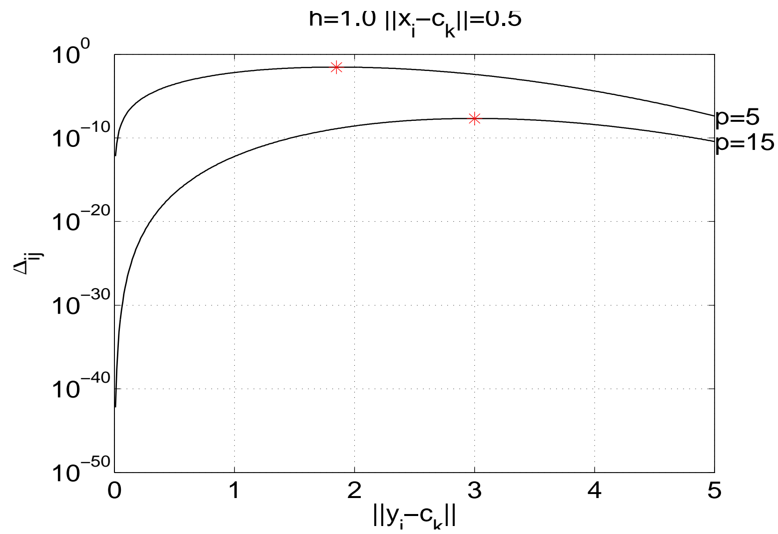
$$\|y_j - c_k\| = \|y_j - c_k\|_* = \frac{\|x_i - c_k\| + \sqrt{\|x_i - c_k\|^2 + 2p_i h^2}}{2}. \tag{2.41}$$

Hence we choose p_i such that,

$$|\Delta_{ij}| \big|_{\|y_j - c_k\| = \|y_j - c_k\|_*} \leq |q_i|\epsilon. \tag{2.42}$$



(a)



(b)

Figure 2.5: The error at y_j due to source x_i, i.e., Δ_{ij} [Equation (2.40)] as a function of $\|y_j - c_k\|$ for different values of p and for (a) $h = 0.5$ and (b) $h = 1.0$. The error increases as a function of $\|y_j - c_k\|$, reaches a maximum and then starts decreasing. The maximum is marked as '*'. $q_i = 1$ and $\|x_i - c_k\| = 0.5$.

In case $\|y_j - c_k\|_* > r_y^k$ we need to choose p_i based on r_y^k, since Δ_{ij} will be much lower there. Hence our strategy for choosing p_i is,

$$|\Delta_{ij}| \Big|_{\left[\|y_j - c_k\| = \min\left(\|y_j - c_k\|_*, r_y^k\right)\right]} \leq |q_i|\epsilon. \tag{2.43}$$

Figure 2.6(a) shows Δ_{ij} as a function of $\|x_i - c_k\|$ for different values of p, $h = 0.4$ and $\|y_j - c_k\| = \min\left(\|y_j - c_k\|_*, r_y^k\right)$. Figure 2.6(b) shows the truncation number p_i required to achieve an error of $\epsilon = 10^{-3}$.

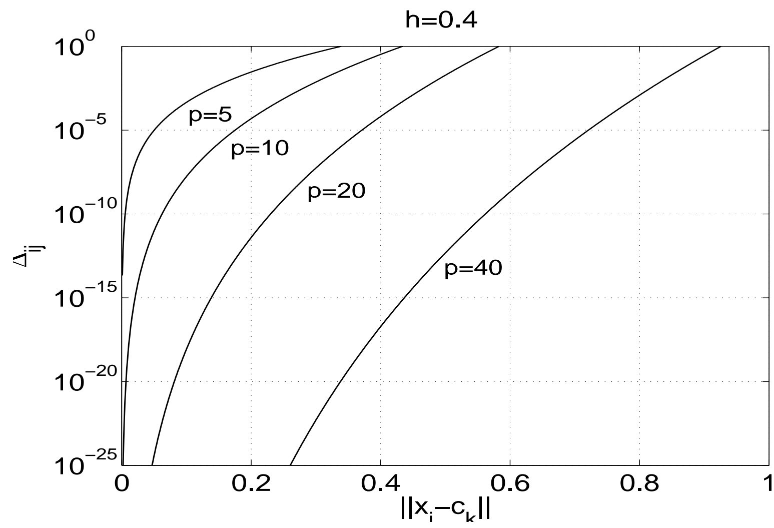
2.6.3 Automatically choosing the number of clusters

Our strategy for choosing the number of clusters is optimized for a uniform distribution of the source points in a unit hypercube. The total computational complexity assuming $M = N$ is $\mathcal{O}(cN)$. The constant term is given by

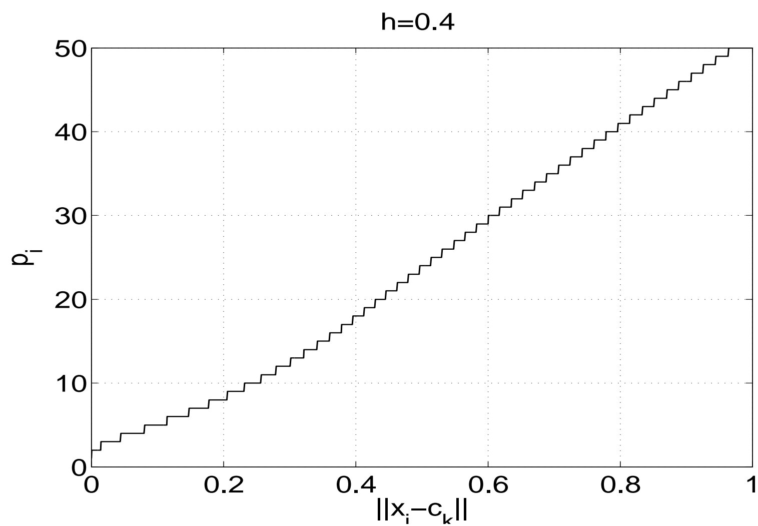
$$c = dK + d\log K + (1+n)r_{(p_{max}-1)d}. \tag{2.44}$$

The truncation number p_{max} and the number of influential clusters n are both functions of K. We choose the number of clusters K for which c is minimum. The truncation number p_{max} is a function of the maximum cluster radius r_x, implicitly via Equation (2.43). If the source and the target points are uniformly distributed in a unit hypercube then a good approximation to the maximum cluster radius would be $r_x \sim K^{-1/d}$. [3] The number of influential neighbor clusters is roughly $n \sim (r/r_x)^d$, where $r = h\sqrt{\ln(1/\epsilon)}$ is the cutoff radius.

[3] If the data lies on a lower dimensional manifold, as usually is the case for structured data in high dimensions, we use the relation $r_x \sim K^{-1/d_{eff}}$. d_{eff} is the actual intrinsic dimensionality of the data.



(a)



(b)

Figure 2.6: The error at y_j due to a source at x_i, i.e., Δ_{ij} [Equation (2.40)] as a function of $\|x_i - c_k\|$ for different values of p, $h = 0.4$ and $\|y_j - c_k\| = \min\left(\|y_j - c_k\|_*, r_y^k\right)$. (b) The truncation number p_i required to achieve an error of $\epsilon = 10^{-3}$.

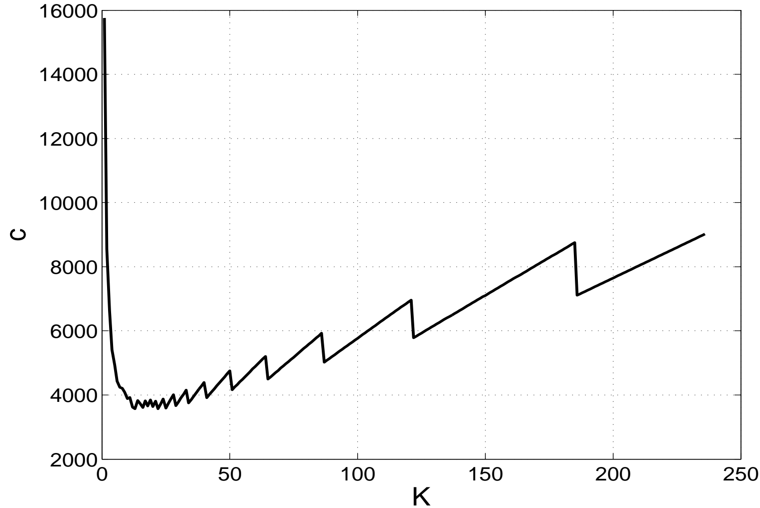
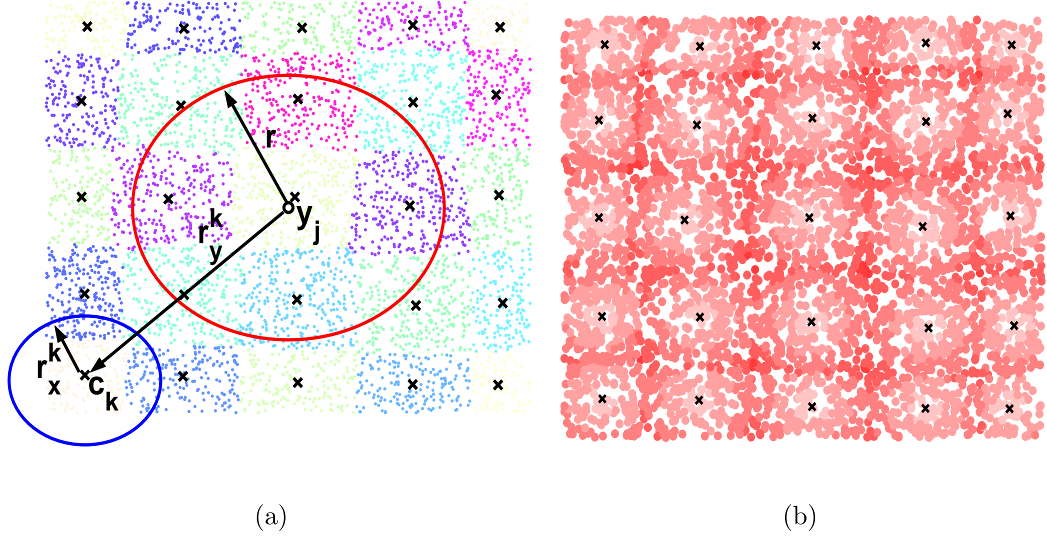


Figure 2.7: The constant c (see Equation (2.44)) as a function of K. $d = 2$, $h = 0.3$, and $\epsilon = 10^{-6}$.

Figure 2.7 shows the constant c as function K. Initially the constant c decreases because as K increases the maximum cluster radius r_x decreases, leading to a smaller truncation number p_{max}. However after a certain point the growth in K dominates the decrease in p_{max}. The optimum K can be found by differentiating Equation (2.44) w.r.t. K and setting it to zero. However since the dependence of p_{max} on K is implicit it is difficult to derive an analytical expression for K. A simple strategy as outlined in Algorithm 1 is to evaluate c for a range of values of K and choose the one for which c is minimum.

2.6.4 Updating the truncation number

We optimized the number of clusters and the maximum truncation number assuming a uniform distribution of source points. However if the data is clustered the truncation number can be smaller. Once we have chosen K we run the farthest



(a) (b)

Figure 2.8: (a) Schematic of the evaluation of the improved fast Gauss transform at the target point y_j. For the source points the results of the farthest point clustering algorithm are shown along with the center of each cluster. A set of coefficients are stored at the center of each cluster. Only the influential clusters within radius r of the target point are used in the final summation. (b) Illustration of the truncation number p_i required for each data point. Dark red color indicates a higher truncation number. Note that for points close to the cluster center the truncation number required is less than that at the boundary of the cluster.

point clustering algorithm which will give us the actual maximum cluster radius. Based on this we can determine the actual maximum truncation number required.

The algorithm is summarized in Algorithm 2 and Figure 2.8(a). Figure 2.8(b) shows the truncation number p_i required for each data point. Dark shade indicates a higher truncation number. Note that for points close to the cluster center the truncation number required is less than that at the boundary of the cluster.

2.7 Nearest Neighbor search based on kd-tree

For small bandwidth h the number of clusters K required can be quite large. The space-subdivision employed by the IFGT algorithm is not hierarchical. As a result nearest influential clusters cannot be searched effectively. When the number of clusters K is large we end up doing a brute force search over all the clusters. Algorithms for efficient computation of nearest neighbors can be incorporated in the algorithm.

ANN (Approximate Nearest Neighbor) is a library written in C++, which supports data structures and algorithms for both exact and approximate nearest neighbor searching in dimensions as high as 20 [51]. The library is based on kd-trees and box-decomposition trees and employs a couple of different search strategies. For our algorithm we build a kd-tree from the K cluster centers. This can be done in $\mathcal{O}(dK \log K)$ time and $\mathcal{O}(dK)$ space. For any target point the influential clusters can be found in $\mathcal{O}(nd \log K)$ time, where n is the number of influential clusters [4].

Sometimes it can happen than the number of clusters chosen K is larger than the number of source points N. In such cases we set $K = N$ and directly evaluate

[4]The exact result is as follows [2]. Given any positive real ϵ, a datapoint p is a $(1+\epsilon)$-approximate nearest neighbor of q if its distance from q is within a factor of $(1+\epsilon)$ of the distance to the true nearest neighbor. With this definition n approximate nearest neighbors can be found in $\mathcal{O}((c_{d,\epsilon} + n) d \log K)$ time where $c_{d,\epsilon} \leq \lceil 1 + 6d/\epsilon \rceil^d$. However in practice the constant is much smaller than the bound. Setting $\epsilon = 0$ will cause the algorithm to compute the exact nearest neighbors but no bound on the running time can be provided. However the algorithm is known to provide significant improvements over brute-force search in dimensions as high as 20.

the contribution from sources within a certain radius r—whos contribution is alteast ϵ–by building a kd-tree directly on the source points.

2.8 FIGTree–Fast Improved Gauss Transform with kd-Tree

So based on the number of clusters K (which depends on the bandwidth h) we have the following three strategies–

1. When the number of clusters is small we use the **Improved Fast Gauss Transform**.
   - Time–$\mathcal{O}(dN \log K + Nr_{(p_{max}-1)d} + Mnr_{(p_{max}-1)d} + dKM)$
   - Space–$\mathcal{O}(Kr_{(p_{max}-1)d} + dN + dM + dK)$
2. When the number of clusters is large we use the **Improved Fast Gauss Transform** along with kd**-tree** on the **cluster centers** to search for the influential clusters.
   - Time–$\mathcal{O}(dN \log K + Nr_{(p_{max}-1)d} + Mnr_{(p_{max}-1)d} + dn \log KM + dK \log K)$
   - Space–$\mathcal{O}(Kr_{(p_{max}-1)d} + dN + dM + dK)$
3. When the number of clusters is almost close the the number of source points we build the kd**-tree** directly on the **source points**.
   - Time–$\mathcal{O}(dN \log N + dn \log NM)$
   - Space–$\mathcal{O}(dN + dM)$

We call the combined algorithm **FIGTree**–**F**ast **I**mproved **G**auss **T**ransform with kd-T**ree**.

2.9 Comparison with the Fast Gauss Transform

We elucidate in detail how our current method differs from the fast Gauss transform. The fast Gauss transform (FGT) [29] is a special case of the more general single level fast multipole method [28], adapted to the Gaussian potential.

The first step of the FGT is the spatial sub-division of the unit hypercube into N_{side}^d boxes of side $\sqrt{2}rh$ where $r < 1/2$ [5]. The sources and targets are assigned to different boxes. Given the sources in one box and the targets in a neighboring box, the computation is performed using one of the following four methods depending on the number of sources and targets in these boxes:

1. Direct evaluation is used if the number of sources and targets are small (in practice a cutoff of the order $\mathcal{O}(p^{d-1})$ is introduced.).

2. If the sources are clustered in a box then they can are transformed into a *Hermite expansion* about the center of the box.

3. This expansion is directly evaluated at each target in the target box if the number of the targets is small.

4. If the targets are clustered then the sources or their expansion are converted to a local *Taylor series* which is then evaluated at each target in the box.

[5] The paper suggests to choose the largest $r \leq 1/2$ such that $N_{side} = 1/\sqrt{2}rh$ is an integer.

Since the Gaussian decays very rapidly only a few neighboring source boxes will have influence on the target box.

2.9.1 Comparison of the IFGT and FGT factorizations

The general fast multipole methods (FMM) [31], of which the FGT is a special case use two kind of expansions of the potential function: the far-field expansion and the local expansion.

For any $x_* \in \mathbf{R}^d$ we call the expansion $\Phi(y, x_i) = \sum_{m=0}^{\infty} b_m(x_i, x_*) S_m(y - x_*)$ *far field* expansion outside a sphere $B^{>}_{R_*}(x_*) = \{y \in \mathbf{R}^d : \|y - x_*\| > R_*\}$, if the series converges for all $y \in B^{>}_{R_*}(x_*)$.

For any $x_* \in \mathbf{R}^d$ we call the expansion $\Phi(y, x_i) = \sum_{m=0}^{\infty} a_m(x_i, x_*) R_m(y - x_*)$ *regular (local)* inside a sphere $B^{<}_{r_*}(x_*) = \{y \in \mathbf{R}^d : \|y - x_*\| < r_*\}$, if the series converges for all $y \in B^{<}_{r_*}(x_*)$.

If the potential has a singular point x_i (unlike a Gaussian), then we use the local expansion for all $\|y - x_*\| < \|x_i - x_*\|$, and far-field expansion for all $\|y - x_*\| > \|x_i - x_*\|$.

The FGT uses the Hermite expansion of the Gaussian for the far-field and the Taylor expansion of the Gaussian (which is obtained by interchanging y and x_i in the Hermite expansion) as the local expansion. The following are the two expansions used by the FGT.

$$e^{-\|y-x_i\|^2/h^2} = \sum_{\alpha \geq 0} \left[\frac{1}{\alpha!} \left(\frac{x_i - x_*}{h} \right)^{\alpha} \right] h_{\alpha} \left(\frac{y - x_*}{h} \right) \quad \text{[far-field Hermite expansion]}, \tag{2.45}$$

$$e^{-\|y-x_i\|^2/h^2} = \sum_{\beta \geq 0} \left[\frac{1}{\beta!} h_\beta \left(\frac{x_i - x_*}{h} \right) \right] \left(\frac{y - x_*}{h} \right)^\beta \text{ [local Taylor expansion]}, \quad (2.46)$$

where $h_\alpha(y)$ are the multivariate Hermite functions [29]. The real benefit of FMM is for singular potential functions whose forces are long ranged and locally non-smooth, hence it is necessary to make use of the tree data structures, local expansions, far-field expansions and translation operators between representations. Translation between local and far-field representations is expensive, but unavoidable in the case of singular potential functions.

The Gaussian is a *regular potential*. For the IFGT we represent the Gaussian as a product of two Gaussians and an exponential (Equation (3.3)), and then use one factorization for the exponential using the Taylor series. The factorization used by the IFGT can be written as follows.

$$e^{-\|y-x_i\|^2/h^2} = \sum_{|\alpha| \geq 0} \left[\frac{2^\alpha}{\alpha!} e^{-\|x_i-x_*\|^2/h^2} \left(\frac{x_i - x_*}{h} \right)^\alpha \right] e^{-\|y_j-x_*\|^2/h^2} \left(\frac{y_j - x_*}{h} \right)^\alpha . \quad (2.47)$$

This factorization has the property that it is both a good far-field and and local expansion, and in fact does a very good job in the whole domain. Thereby it avoids the need for two different representations and the expensive translation operation.

Figure 2.9 shows the absolute value of the actual error between the one dimensional Gaussian ($e^{-(x_i-y)/h^2}$) and the different series approximations. The Gaussian was centered at $x_i = 0$. All the series were expanded about $x_* = 1.0$. $p = 5$ terms were retained in the series approximation. From the plot it can be seen that the Hermite expansion is essentially a far field expansion which gives better approximation as we move far away from x_*. The Taylor expansion of the Gaussian is a local expansion giving good approximation only for a region very close to x_*. *The*

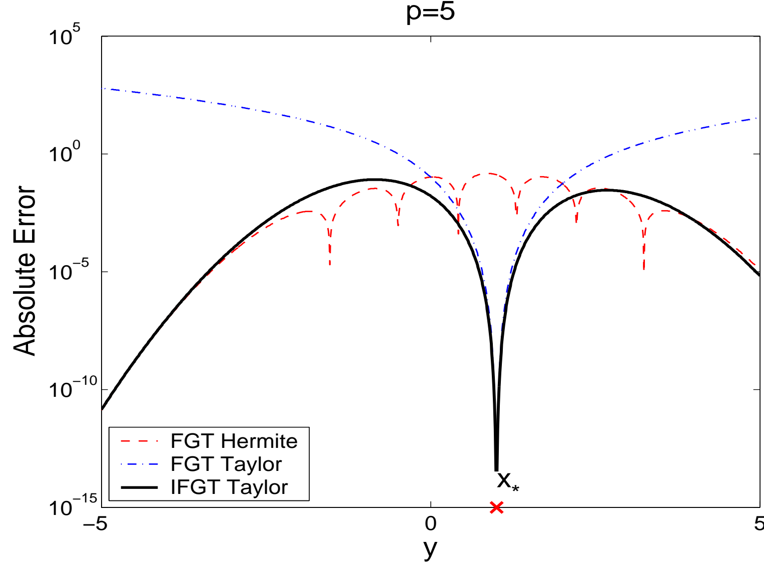


Figure 2.9: The absolute value of the actual error between the one dimensional Gaussian ($e^{-(x_i-y)/h^2}$) and different series approximations. The Gaussian was centered at $x_i = 0$ and $h = 1.0$. All the series were expanded about $x_* = 1.0$. $p = 5$ terms were retained in the series approximation.

Taylor expansion used by the IFGT can serve both as the far field as well as the local expansion.

2.9.2 Translation

Since the original FGT uses two representations it must convert between them using a process called *translation*. The original FGT in d dimensions represents the solution using p^d coefficients. The cost of translation is $\mathcal{O}(dp^{d+1}(2n+1)^d min((\sqrt{2}rh)^{-d/2}, M))$. The new version of the FGT proposed in [30] reduces the cost of translating the Hermite series. The new version is based on replacing the Hermite and Taylor expansions with an expansion in terms of exponentials (plane

waves). Because of this, the translation operator becomes diagonal. This reduces the cost of translation from $\mathcal{O}(d(2n+1)^d p^{d+1})$ to $\mathcal{O}(3dp^d)$. In any case the cost of translation grows exponentially with dimension [6]. In contrast our method uses just one representation with the property that it is both a good far-field and and local expansion (See Figure 2.9). Thereby it avoids the need for two different representations and the expensive translation operation.

2.9.3 Error bounds

In this section we compare the number of terms needed to achieve a desired error for the truncated expansions used by the FGT and the IFGT algorithm. We cannot compare both the expressions in terms of p since the truncation method is different for the FGT and the IFGT. We need to see the *total number of terms* that need to be retained to achieve a given target error. For the Hermite expansion all terms with multi indices $\alpha > p$ are ignored (as a result we retain p^d terms) while in the case of IFGT all terms with multi indices of degree $|\alpha| > p$ are ignored (as a result we retain all monomials with degree $\leq p-1$ (*i.e.* a total of $r_{(p-1)d}$ terms)).

Let $|x_i(j) - x_*(j)| = a$ and $|y(j) - x_*(j)| = b$. The error due to truncation in IFGT after ignoring all terms with multi indices of degree $|\alpha| > p$ can be bounded as follows.

$$\begin{aligned} |error_{IFGT}| &< \frac{2^p}{p!}\left(\frac{\|x_i - c_k\|}{h}\right)^p \left(\frac{\|y_j - c_k\|}{h}\right)^p e^{-(\|x_i - c_k\| - \|y_j - c_k\|)^2/h^2}, \\ &= \frac{2^p}{p!}\left(\frac{dab}{h^2}\right)^p e^{-d(a-b)^2/h^2}. \end{aligned}$$

[6] Also the details of the scheme are presented only for $d \leq 3$.

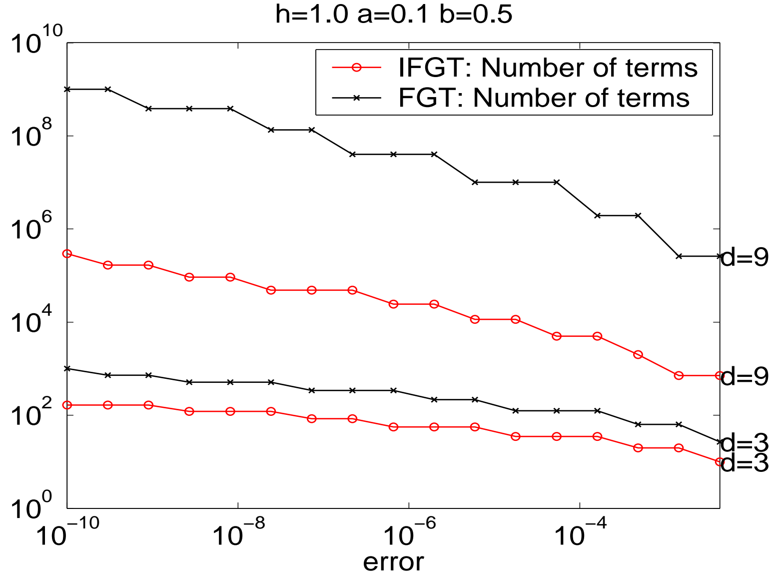


Figure 2.10: The total number of terms required by the IFGT and the FGT series expansions to achieve a desired error bound.

The error due to truncation of either the Hermite series or the Taylor series in FGT after ignoring all terms with multi-indices $\alpha > p$ can be bounded as follows (This bound can be derived using the approach detailed in [3]).

$$|error_{FGT}| < \frac{e^{-db^2/2h^2}}{(1-r)^d} \sum_{k=0}^{d-1} \binom{d}{k} (1-r^p)^k \left(\frac{r^p}{\sqrt{p!}}\right)^{d-k}.$$

where $r = \sqrt{2}a/h$. Figure 2.10 compares the total number of terms required by the IFGT and the FGT series expansions to achieve a desired error bound. Our expansion and truncation scheme results in a substantial reduction in the number of terms.

2.9.4 Spatial data structures

The original FGT uses boxes to subdivide the space. However such a simple space subdivision scheme is not suitable for high dimensions. If each dimension

of a unit hyper cube is divided into N_{side} parts, then the number of boxes grows exponentially with dimension as N_{side}^d, resulting in prohibitive memory requirements. In most statistical and machine learning applications we do not have truly high dimensional data. The data will typically lie on low dimensional manifolds. The consequence of this is that most of the boxes will be empty and we will be spending resources in searching nonempty neighboring boxes. To adaptively fit the density of points, the IFGT uses the farthest-point algorithm to subdivide the space. Table 2.2 compares the number of boxes required by the FGT and the number of clusters required by the IFGT as a function of the data dimensionality d. For example in a seven dimensional space while the FGT subdivides the space into 2187 boxes the IFGT just needs 67 clusters.

2.9.5 Exponential growth of complexity with dimension

The total computational complexity of the FGT is of the form [29]

$$O(p^d N) + O(p^d M) + O(dp^{d+1}(2n+1)^d min((\sqrt{2}rh)^{-d/2}, M)). \tag{2.48}$$

The third term $dp^{d+1}(2n+1)^d min((\sqrt{2}rh)^{-d/2}, M)$ is essentially a constant depending on the number of box-box interactions and the cost of translating a Hermite expansion into a Taylor series. The translation is one of the most expensive step in any FMM algorithm. Even though it does not depend on N, the constant term grows exponentially with increasing dimensionality. Table. 2.2 shows the constant term as a function of d for $h = 0.5$ and $\epsilon = 10^{-6}$. This suggests that the FGT may not be practical for dimensions > 3. Also the constant term p^d grows exponentially

with dimension. Compare this with the computational complexity of the IFGT.

$$\mathcal{O}(N \log K + N r_{(p_{max}-1)d} + M n r_{(p_{max}-1)d} + KM). \tag{2.49}$$

Assuming $M = N$, the complexity is $\mathcal{O}(\left[K + \log K + (1+n) r_{(p-1)d}\right] N)$. The constant $r_{(p-1)d}$ is asymptotically polynomial in d. For $d \to \infty$ and moderate p, the number of terms is $\mathcal{O}(d^p)$. Since we cluster only the source points. We do not use any expensive translation operation. Table 2.2 compares the number of terms required for FGT (p^d) with the number of terms required by IFGT ($r_{(p-1)d}$).

2.10 Numerical Experiments

In this section we present numerical studies of the speedup and error as a function of

- the number of data points, N,
- the data dimensionality, d,
- the bandwidth h of the Gaussian kernel,
- the desired error, ϵ,
- and the distribution of the source and target points.

We compare the following four methods–

- Direct–Naive $\mathcal{O}(N^2)$ implementation of the Gauss transform.

- FGT [7]–The fast Gauss Transform as described in [29].
- FIGTree [8]–The proposed improved fast Gauss transform along with the *kd*-tree based nearest neighbor search.
- Dual-tree method [9]–The *kd*-tree based dual-tree algorithm of [26].

All the algorithms were programmed in C++ or C with MATLAB bindings and were run on a 1.83 GHz Pentium-M processor with 1 GB of RAM.

2.10.1 Speedup as a function of N

We first study the performance as the function of N for $d = 3$. N points were uniformly distributed in a unit cube. The Gauss transform was evaluated at $M = N$ points uniformly distributed in the unit cube. The weights q_i were uniformly distributed between 0 and 1. The parameters for the algorithms were automatically chosen without any user intervention. The target error was set to 10^{-3}.

Figure 2.11 shows the results for all the various methods as a function of N for bandwidth $h = 0.25$. The following observations can be made–

- As expected, for FIGTree the computational cost grows linearly with N.

[7]The code for FGT for $d \leq 3$ was downloaded from the website `http://www.cs.ubc.ca/~awll/nbody_methods.html`. For higher dimensions we wrote our own code.

[8]The code for the FIGTree implementation is available at `http://www.umiacs.umd.edu/~vikas/Software/IFGT/IFGT_code.htm`. The ANN library (`http://www.cs.umd.edu/~mount/ANN/`) was used to build the *kd*-tree and perform the nearest neighbor search.

[9]The code for the dual-tree algorithms was downloaded from the website `http://www.cs.ubc.ca/~awll/nbody_methods.html`.

- For the FGT the cost grows linearly only after a large N when the linear term $\mathcal{O}(p^d N)$ dominates the initial large fixed cost of Hermite-Taylor translation. A similar jump in the performance can also be seen in the original FGT paper (See Tables 2 and 4 in [29]). While the computational complexity of FIGTree grows linearly with N, the linear growth of FGT is a bit intricate because of the various cutoffs involved and the cost of the translation. In order to understand the complexity of the FGT, we refer to Figure 2.12 where we plot the theoretical complexity as a function of N for $d = 3$. Initially before the translation has kicked in (*i.e.* before point A in Figure 2.12) the growth is linear in N. The sudden jump observed is due to the constant associated with the high cost of translation after which the growth is dominated by the constant term. From this point all box-box interactions are performed only by the translations. However after a large N the asymptotics dominate because the growth in N has dominated the cost of translation (*i.e.* after point B in Figure 2.12). In practice especially for high dimensions the constant term due to translation is so large that the N has to be typically very large for the asymptotic performance to kick in.

- FIGTree shows a better speedup than the FGT. However the FGT finally catches up with FIGTree (*i.e.* the asymptotic performance starts dominating) and shows a speedup similar to that of the FIGTree. However this happens typically after a very large N. This value of N increases with the dimensionality of the problem.

- With regard to the actual error the FIGTree error is closer to the target than is the FGT. The dual-tree algorithm shows the best performance in this regard, and was very close to the target error.

- The dual-tree algorithm appears to be doing $\mathcal{O}(N^2)$ work. Also it takes much more time than the direct evaluation probably because of the time taken to build up the kd-trees. Dual-tree algorithms show good speedups only at very small bandwidths. Figure 2.13 shows the same results for small bandwidth of $h = 0.05$. However the FIGTree shows better speedup than the dual-tree method for these cases. For small bandwidths FIGTree leverages its speedup by using the ANN library for efficient neighbor search.

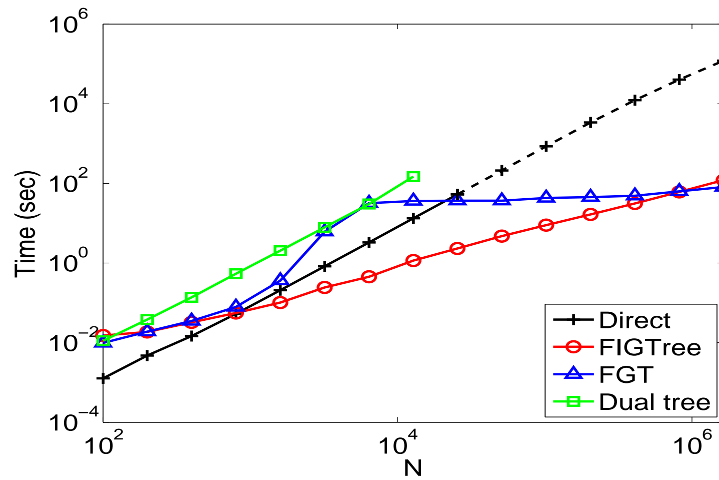
- Figure 2.14 shows the same results for large bandwidth of $h = 1.0$. For large bandwidth both the FGT and FIGTree showed very similar performance.

2.10.2 Speedup as a function of d

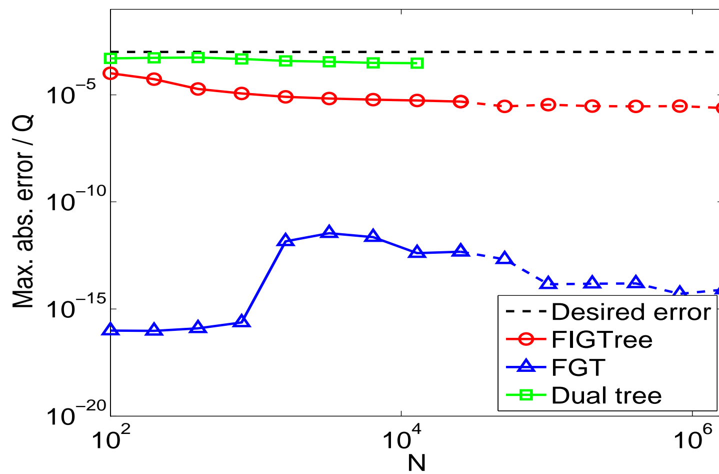
The main advantage of FIGTree is in higher dimensions where we can no longer run the FGT algorithm. Figure 2.15 shows the performance for a fixed $N = M = 50,000$ as a function of d for a fixed bandwidth of $h = 1.0$.

- FGT becomes impractical after three dimensions with the cost of translation increasing with dimensionality. The FGT gave good speedup only for $d \leq 4$.

- For FIGTree, as d increases the crossover point (*i.e.*, the N after which FIGTree shows a better performance than the direct) increases. Since the



(a)



(b)

Figure 2.11: *Scaling with* N *for* $d = 3$ *and* $h = 0.25$. (a) The running times in seconds and (b) the maximum absolute error relative to the total weight Q for direct evaluation, FGT, dual tree, and FIGTree. The target error was set to 10^{-3}. The bandwidth was $h = 0.25$. The source and target points were uniformly distributed in a unit cube. The weights q_i were uniformly distributed between 0 and 1. For $N > 25600$ the timing results for the direct evaluation were obtained by evaluating the Gauss transform at $M = 100$ points and then extrapolating the results. The dual-tree algorithm could not be run after a certain N due to limited memory.

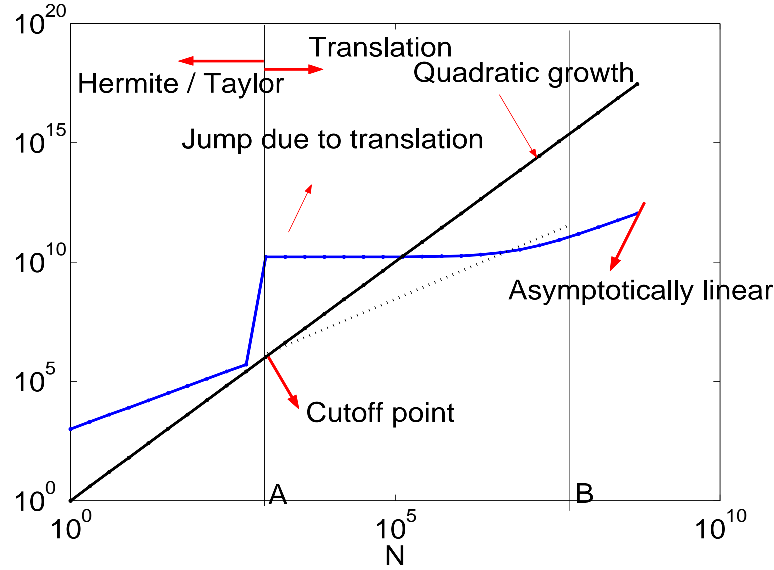
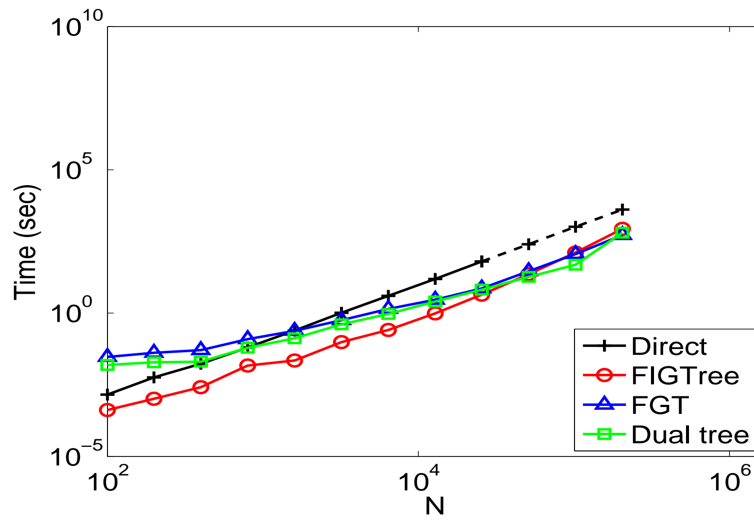


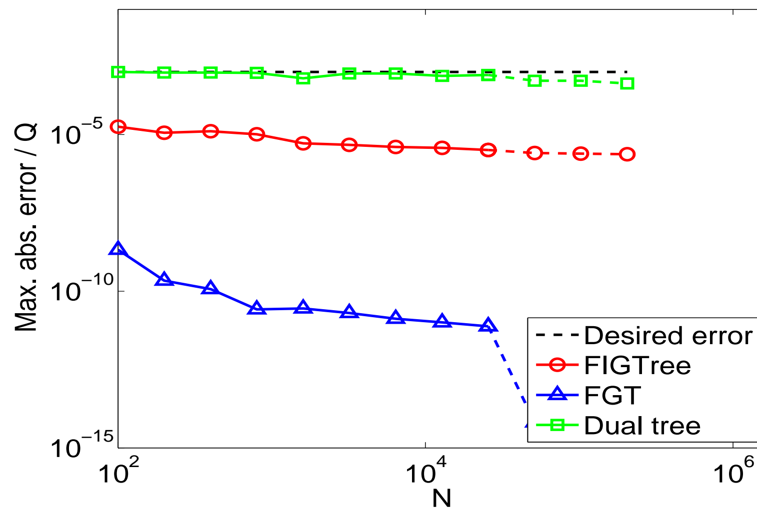
Figure 2.12: The theoretical complexity $(\mathcal{O}(2p^dN + dp^{d+1}(2n+1)^d min((\sqrt{2}rh)^{-d/2}, N)))$ of the FGT showing different regions as a function of $N = M$ for $d = 3$.

crossover point increases with d for $N = M = 50,000$ we were able to achieve good speedups till $d = 10$. The dual-tree method could not be run for the bandwidth chosen.

- Figure 2.16 shows the performance for a fixed $N = M = 20,000$ as a function of d. In this case for each dimension we set the bandwidth $h = 0.5\sqrt{d}$ (Note that $\sqrt{d}$ is the length of the diagonal of a unit hypercube). The bandwidth of this order is sometimes used in high dimensional data in a lot of machine learning tasks and good generalization performance has been achieved. With h varying with dimension we were able to run the algorithm for arbitrary high dimensions. The dual-tree algorithm took more than the direct method for such bandwidths.

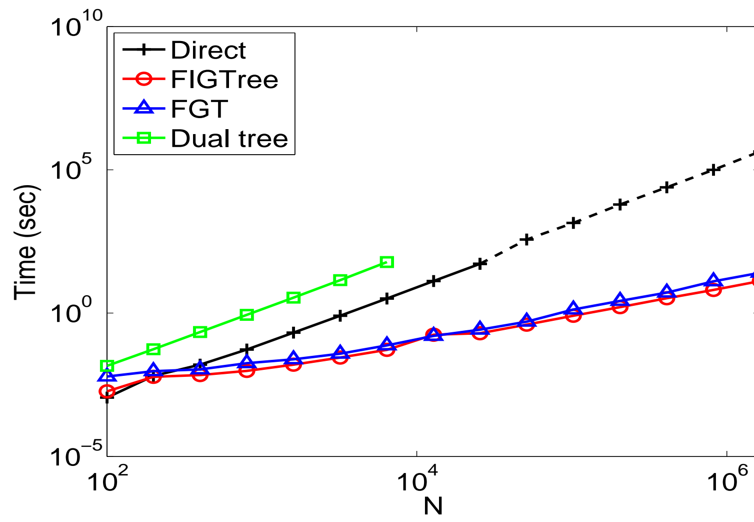


(a)

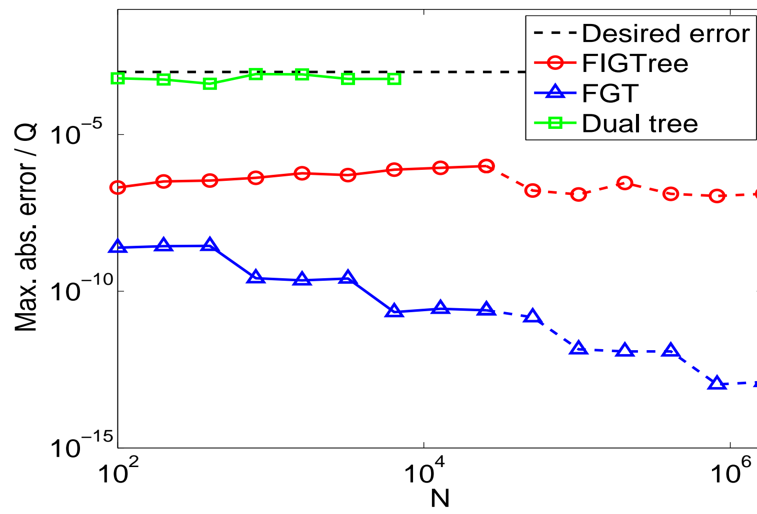


(b)

Figure 2.13: *Scaling with N for $d = 3$ and $h = 0.05$.* Same as Figure 2.11 with $h = 0.05$.



(a)



(b)

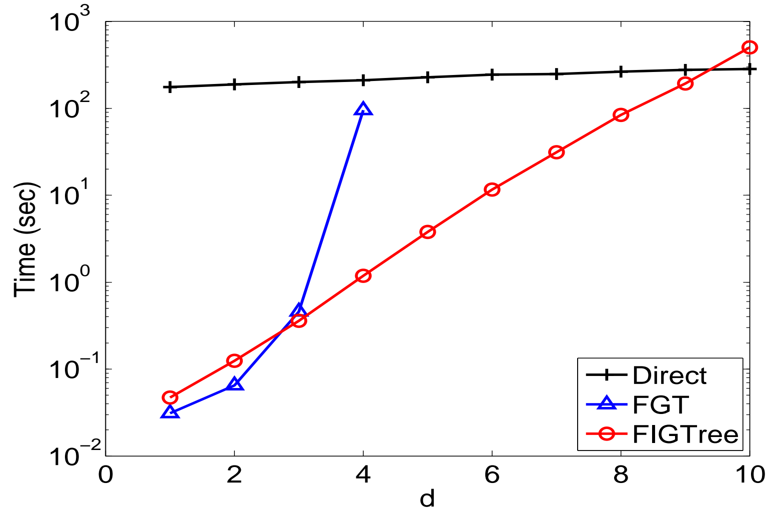
Figure 2.14: *Scaling with N for $d = 3$ and $h = 1.0$.* Same as Figure 2.11 with $h = 1.0$.

- Figure 2.17 shows the performance for a fixed $N = M = 10,000$ as a function of d for a fixed small bandwidth of $h = 0.001$. FIGTree shows much better speedups than the dual-tree algorithms. For FIGTree the number of clusters was almost close the the number of source points and hence we build the *kd*-tree directly on the source points. The error was almost zero for the FIGTree.

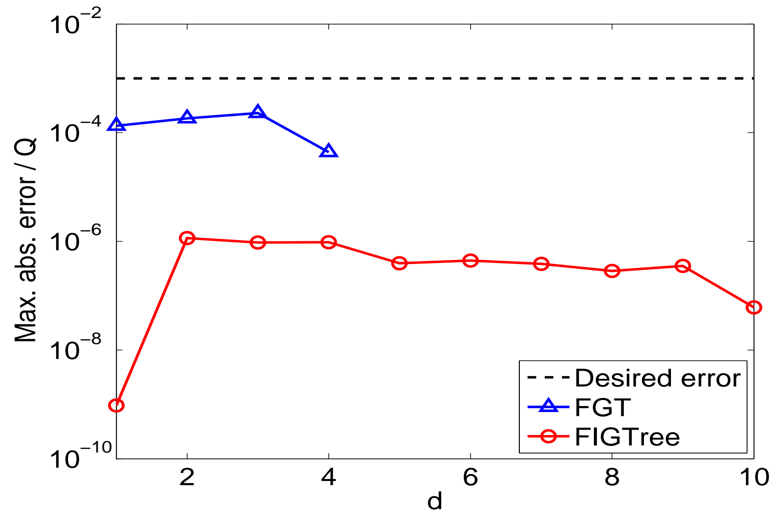
2.10.3 Speedup as a function of the bandwidth h

One of the important concerns for N-body algorithms is their *scalability with bandwidth h*. Figure 2.18 shows the performance of the FIGTree and the dual-tree algorithm as a function of the bandwidth h. The other parameters were fixed at $N = M = 7,000$, $\epsilon = 10^{-3}$, and $d = 2, 3, 4, \text{and}, 5$.

- The dual-tree algorithm shows good speedup only at small bandwidths. At large bandwidths the dual-tree algorithms ends up doing the same amount of work as the direct implementation. The dual-tree appears to take larger time than the direct probably because of the time taken to build up the *kd*-trees.
- The dual-tree algorithm also shows good speedup at very large bandwidths for small dimensions (See the curve for $d = 2$ in Figure 2.18(a)).
- For large bandwidths FIGTree shows better speedups than the dual-tree algorithm.
- FIGTree also performs better than the dual-tree for small bandwidths. For small bandwidths the number of clusters K is quite large. When the number

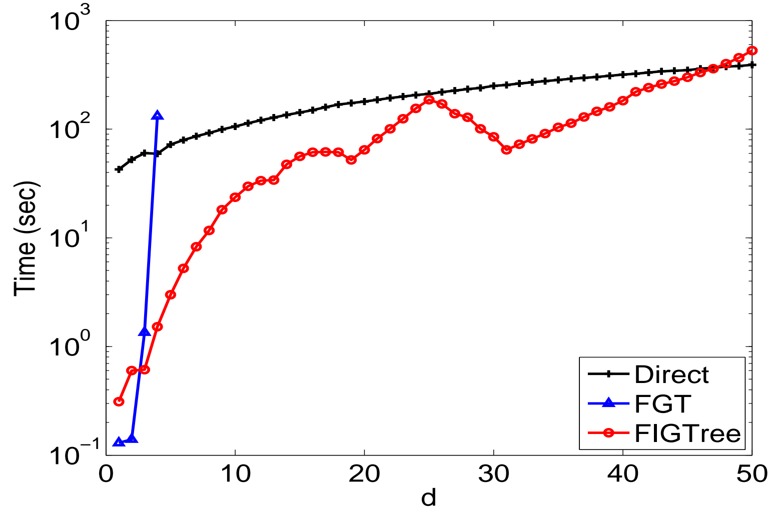


(a)

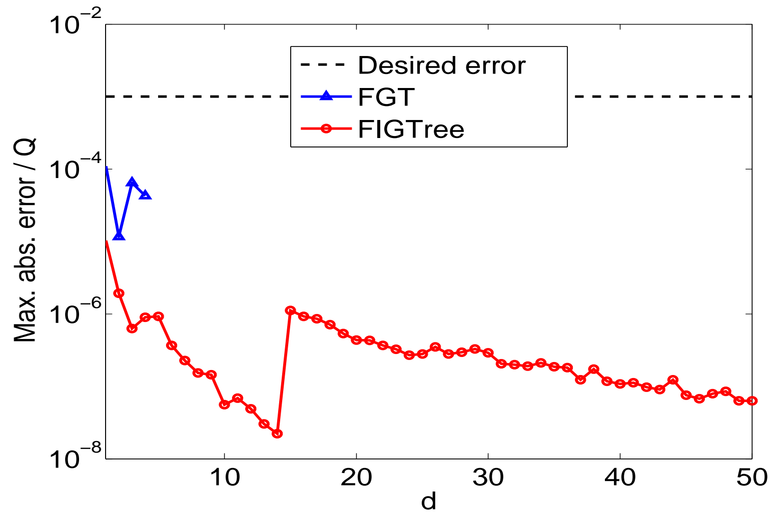


(b)

Figure 2.15: *Scaling with d for $h = 1.0$.* (a) The running times in seconds and (b) the maximum absolute error relative to the total weight Q for direct evaluation, FGT, and FIGTree as a function of the dimension d. The target error was set at $\epsilon = 10^{-3}$. The bandwidth was $h = 1.0$. $N = 50,000$ source and target points were uniformly distributed in a unit hyper cube. The weights q_i were uniformly distributed between 0 and 1.

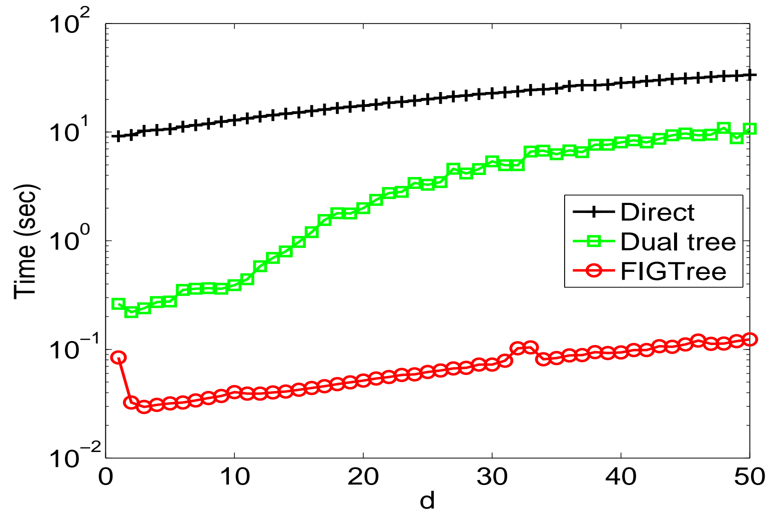


(a)

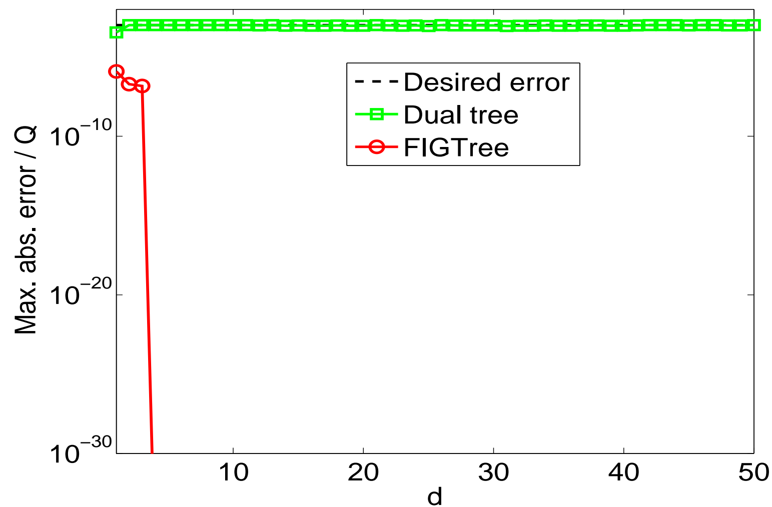


(b)

Figure 2.16: *Effect of bandwidth* $h = 0.5\sqrt{d}$.(a) The running times in seconds and (b) the maximum absolute error relative to the total weight Q for direct evaluation, FGT, and FIGTRee as a function of the dimension d. The target error was set at $\epsilon = 10^{-3}$. The bandwidth was $h = 0.5\sqrt{d}$. $N = 20,000$ source and target points were uniformly distributed in a unit hyper cube. The weights q_i were uniformly distributed between 0 and 1.



(a)



(b)

Figure 2.17: *Scaling with d for $h = 0.001$.* (a) The running times in seconds and (b) the maximum absolute error relative to the total weight Q for direct evaluation, FGT, and FIGTRee as a function of the dimension d. The target error was set at $\epsilon = 10^{-3}$. The bandwidth was $h = 0.001$. $N = 10,000$ source and target points were uniformly distributed in a unit hyper cube. The weights q_i were uniformly distributed between 0 and 1.

of clusters is large we use IFGT along with kd-tree on the cluster centers to search for the influential clusters.

2.10.4 Speedup as a function of the desired error ϵ

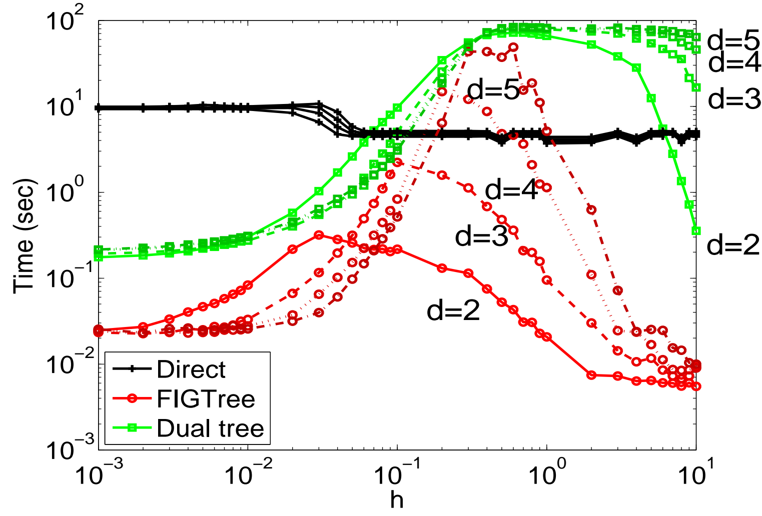
Figure 2.19 shows the tradeoff between the time taken and the desired error ϵ.

- A decrease in running time is obtained at the expense of reduced precision for both FIGTtree and the dual-tree method.
- Also the true error for FIGTree is below the desired, thus validating the error bound and the choice of the parameters.
- The error bound for FIGTree is not tight compared to that of the dual-tree. The plot demonstrates that for FIGTree there is a further scope for improvement by changing the parameters chosen.

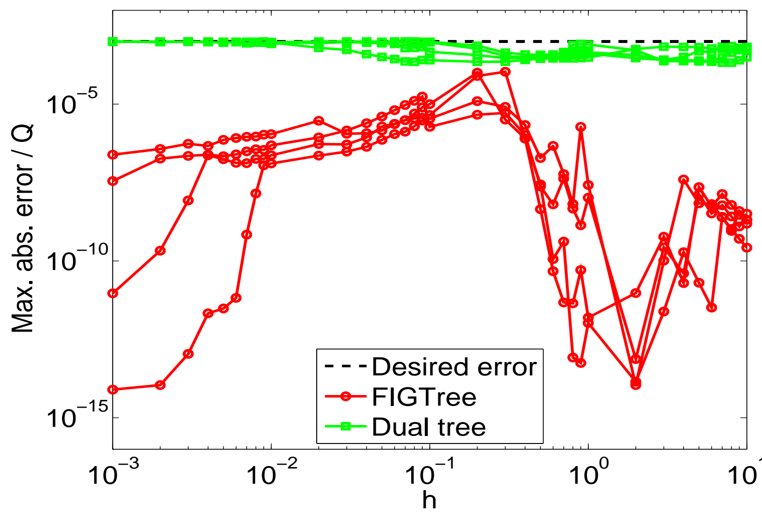
2.10.5 Structured data

Until now we showed results for the worst case scenario–data uniformly distributed in a unit hypercube. However if there is structure in the data, *i.e.*, the data is either clustered or lie on some smooth lower dimensional manifold, then the algorithms show much better speed up. Figure 2.20 compares the time taken by the FIGTree and dual tree methods as a function of h for four different scenarios:

1. Case 1: Both source and target points are uniformly distributed.

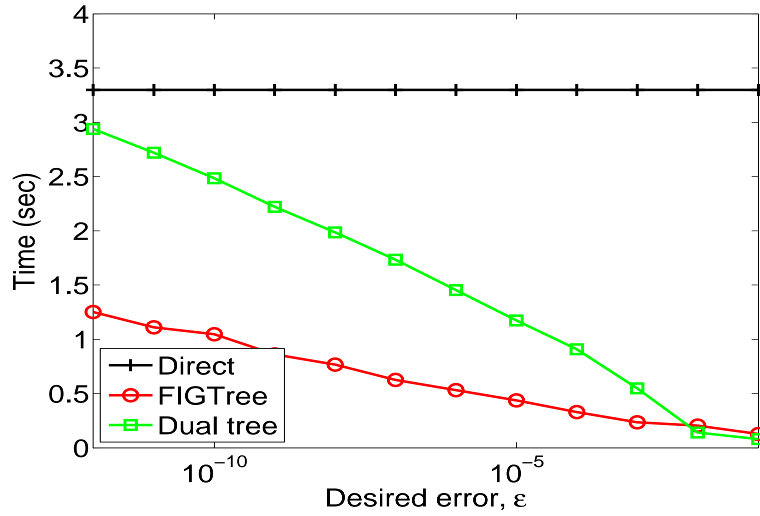


(a)

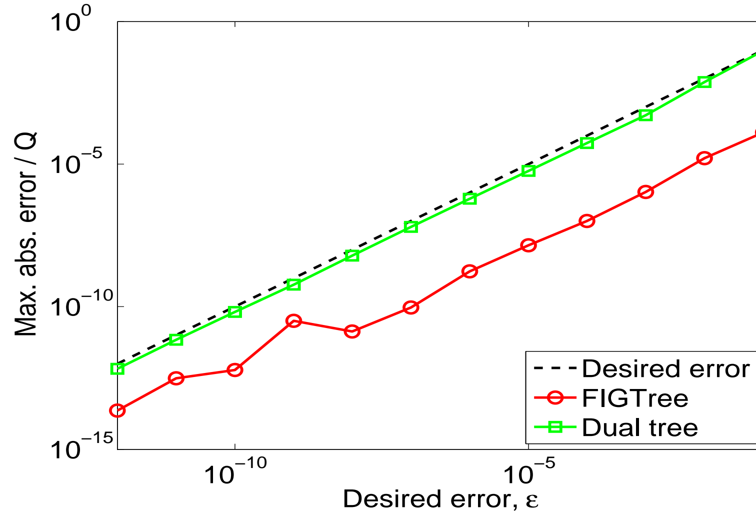


(b)

Figure 2.18: *Effect of bandwidth h.* (a) The running times in seconds and (b) the maximum absolute error relative to the total weight Q for direct evaluation, IFGT, and the kd-tree dual-tree algorithm as a function of the bandwidth h. The target error was set at $\epsilon = 10^{-3}$. $N = 7,000$ source and target points were uniformly distributed in a unit hyper cube of dimension $d = 2, 3, 4,$ and, 5. The weights q_i were uniformly distributed between 0 and 1.



(a)



(b)

Figure 2.19: *Effect of the desired error* ϵ. (a) The running times in seconds and (b) the maximum absolute error relative to the total weight Q for the dual-tree and the FIGTree as a function of the desired error ϵ. The bandwidth was $h = 0.03$. $N = 5,000$ source and target points were uniformly distributed in a unit hyper cube of dimension $d = 2$. The weights q_i were uniformly distributed between 0 and 1.

2. Case 2: Source points are clumpy while the target points are uniformly distributed.

3. Case 3: Source points uniformly distributed while the target points are clumpy.

4. Case 4: Both source and target points are clumpy.

The clumpy data was generated from a mixture of 10 Gaussians. The following observations can be made:

- For the dual tree method clumpiness either in source or target points gives better speedups. However, the performance is still worse than FIGTree.

- For the FIGTree clumpiness in source points gives a much better speed up than uniform distribution. Clumpiness in target points does not matter since FIGTree clusters only the source points.

2.10.6 Summary

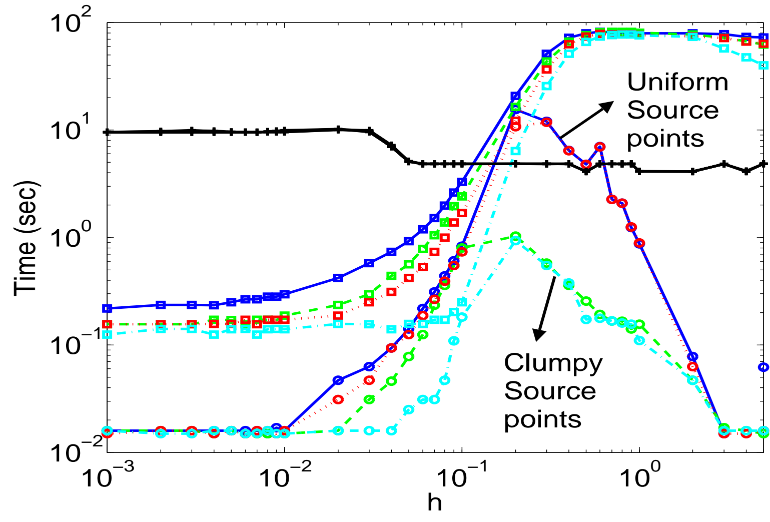
Table 2.3 summarizes the conditions under with various algorithms perform better.

- Dual-tree algorithms give good speedups only for small bandwidths.

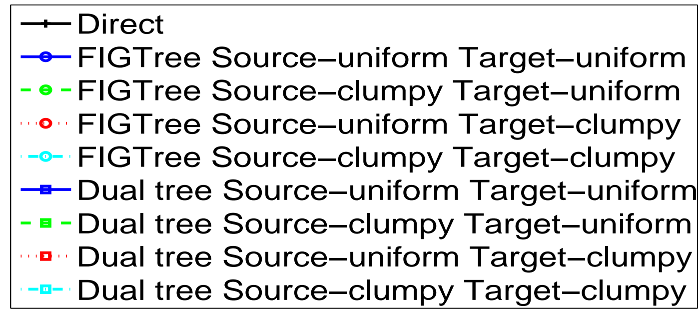
- FGT performs well only for $d \leq 3$.

- For large bandwidths the IFGT is substantially faster than the other methods.

- The FIGTRee algorithm–IFGT combined with *kd*-tree based nearest neighbor search–gives the best speedups for small bandwidths.



(a)



(b)

Figure 2.20: *Effect of clumpy data.* The running times for the different methods as a function of the bandwidth h. [$\epsilon = 10^{-3}$, $N = M = 7,000$, and $d = 4$]

- In addition, for moderate bandwidths and moderate dimensions IFGT performs better than dual-tree algorithms.
- For moderate bandwidths and large dimensions we still have to resort to direct summation.

2.11 Applications

The proposed algorithm can be used in any scenario where we encounter sums of Gaussians. Applications include kernel density estimation [96], prediction in SVMs [13], and mean prediction in Gaussian process regression [90]. For training, the algorithm can be embedded in a conjugate-gradient or any other suitable optimization procedure [95, 73]. In some unsupervised learning tasks the algorithm can be embedded in iterative methods used to compute the eigen vectors [15]. While providing experiments for all applications is beyond the scope of this theis, we demonstrate the use of FIGTree to accelerate the following two learning tasks:

1. *Multivariate kernel density estimation* (Chapter 5) This involves a direct application of FIGTree. All the weights q_i are positive and the bandwidth of the Gaussian kernel decreases as the number of points increases.
2. *Gaussian process regression* (Chapter 7) For this task FIGTree is embedded in a conjugate-gradient procedure. In this case the weights can be either positive or negative. The bandwidth of the Gaussian kernel needed is generally large and is fairly constant with increasing the number of points.

2.12 Conclusions

We proposed the fast improved Gauss transform with *kd*-tree (FIGTree) which is capable of computing the Gauss transform in $\mathcal{O}(M+N)$ time in dimensions as high as tens for moderate bandwidths and as high as hundreds for large and small bandwidths. The reduction is based on a new multivariate Taylor series expansion (which can act both as a local as well as a far field expansion) scheme combined with the efficient space subdivision using the k-center algorithm. We derived tight point-wise error bounds and gave a strategy to choose the parameters of the algorithm. Numerical experiments demonstrated the speedup achieved over the original FGT and the dual tree algorithm.

The following are two simple extensions.

- The bandwidth h is different for each dimension, *i.e.*,

$$G(y_j) = \sum_{i=1}^{N} q_i e^{-(y_j - x_i)^T \Sigma^{-1} (y_j - x_i)} = \sum_{i=1}^{N} q_i e^{-\sum_{k=1}^{d} (y_j(k) - x_i(k))^2 / h_k^2}, \tag{2.50}$$

  where Σ is diagonal matrix with the kth element equal to h_k^2. In this case we can divide each co-ordinate of the source and target points with the corresponding bandwidth h_k and then use the algorithm with bandwidth $h = 1$.

- More generally we can have

$$G(y_j) = \sum_{i=1}^{N} q_i e^{-(y_j - x_i)^T H^{-1} (y_j - x_i)}, \tag{2.51}$$

  where H is a symmetric positive definite $d \times d$ matrix called the bandwidth matrix. In this case we can factorize the inverse bandwidth matrix as $H^{-1} =$

$U^T\Sigma^{-1}U = (\Sigma^{-1/2}U)^T(\Sigma^{-1/2}U)$. Now we can apply the following linear transformation $x \to \Sigma^{-1/2}Ux$ to each of the source and target points and then use the algorithm with $h = 1$. This is equivalent to rotating and scaling the points before using the FIGTree.

We also point out that the algorithm is easily adaptable in an online setting. If a new target point arrives them we just have to sum the contributions from all its influential neighbor clusters. If a new single source point arrives we just add its contribution directly to all the target points. In case a lot of source points arrive in a batch then we update the coefficients of the clusters to which the source points belong to and then reevaluate the contribution at the target points.

The C++ code with MATLAB bindings for the IFGT implementation is available under Lesser GPL at `http://www.umiacs.umd.edu/~vikas/Software/IFGT/IFGT_code.htm`.

Our experimental results indicate that it easy to get good speedups at very large or very small bandwidths. For moderate bandwidths and moderate dimensions the improved fast Gauss transform is capable of giving good speedups. Getting good speedups for moderate bandwidths and large dimensions remains an open research problem.

One of the goals when designing these kind of algorithms is to give high accuracy guarantees. But sometimes because of the loose error bounds we end up doing much more work than necessary. In such a cause the proposed algorithm can be used by just choosing a truncation number p and seeing how the algorithm performs.

Input : d (dimension)

h (bandwidth)

ϵ (error)

N (number of sources)

Output: K (number of clusters)

r (cutoff radius)

p_{max} (maximum truncation number)

Define

$\delta(p,a,b) = \frac{1}{p!}\left(\frac{2ab}{h^2}\right)^p e^{-(a-b)^2/h^2}$, $b_*(a,p) = \frac{a+\sqrt{a^2+2ph^2}}{2}$, and $r_{pd} = \binom{p-1+d}{d}$;

Choose the cutoff radius $r \leftarrow \min(\sqrt{d}, h\sqrt{\ln(1/\epsilon)})$;

Choose $K_{limit} \leftarrow \min\left(\lceil 20\sqrt{d}/h \rceil, N\right)$ (a rough bound on K);

for $k \leftarrow 1 : K_{limit}$ **do**

compute an estimate of the maximum cluster radius as $r_x \leftarrow k^{-1/d}$;

compute an estimate of the number of neighbors as $n \leftarrow \min\left((r/r_x)^d, k\right)$;

choose $p[k]$ such that $\delta(p = p[k], a = r_x, b = \min[b_*(r_x, p[k]), r + r_x]) \leq \epsilon$;

compute the constant term $c[k] \leftarrow dk + d\log k + (1+n)r_{(p[k]-1)d}$

end

choose $K \leftarrow k_*$ for which $c[k_*]$ is minimum. $p_{max} \leftarrow p[k_*]$.

Algorithm 1: Choosing the parameters for the IFGT

Input :

$x_i \in \mathbf{R}^d$ $i = 1, \ldots, N$ `/*` N `sources in` d `dimensions. */`

$q_i \in \mathbf{R}$ $i = 1, \ldots, N$ `/* source weights. */`

$h \in \mathbf{R}^+$ $i = 1, \ldots, N$ `/* source bandwidth. */`

$y_j \in \mathbf{R}^d$ $j = 1, \ldots, M$ `/*` M `targets in` d `dimensions. */`

$\epsilon > 0$ `/* Desired error. */`

Output: Computes an approximation $\hat{G}(y_j)$ to $G(y_j) = \sum_{i=1}^{N} q_i e^{-\|y_j - x_i\|^2/h^2}$. such that the $\frac{|\hat{G}(y_j) - G(y_j)|}{Q} \leq \epsilon$, where $Q = \sum_{i=1}^{N} |q_i|$.

Step 0 *Define* $\delta(p, a, b) = \frac{1}{p!} \left(\frac{2ab}{h^2}\right)^p e^{-(a-b)^2/h^2}$ *and* $b_*(a, p) = \frac{a + \sqrt{a^2 + 2ph^2}}{2}$;

Step 1 *Choose the cutoff radius* r*, the number of clusters* K*, and the maximum truncation number* p_{max} *using Algorithm 1*;

Step 2 *Divide the* N *sources into* K *clusters,* $\{S_k\}_{k=1}^{K}$*. Let* c_k *and* r_x^k *be the center and radius respectively of the* k^{th} *cluster. Let* $r_x = \max_k \left(r_x^k\right)$. ;

Step 3 *Update the maximum truncation number based on the actual* r_x*, i.e., choose* p_{max} *such that* $\delta(p = p_{max}, a = r_x, \min\left[b_*(r_x, p_{max}), r + r_x\right]) \leq \epsilon$

Step 4 *For each cluster* S_k *with center* c_k *compute the coefficients* C_α^k.

$C_\alpha^k = \frac{2^\alpha}{\alpha!} \sum_{x_i \in S_k} q_i e^{-\|x_i - c_k\|^2/h^2} \left(\frac{x_i - c_k}{h}\right)^\alpha \mathbf{1}_{|\alpha| \leq p_i - 1} \ \forall |\alpha| \leq p_{max} - 1$

The truncation number p_i *for each source is selected such that*

$\delta(p = p_i, a = \|x_i - c_k\|, \min\left[b_*(\|x_i - c_k\|, p_i), r + r_x^k\right]) \leq \epsilon$;

Step 5 *For each target* y_j *the discrete Gauss transform is evaluated as*

$\hat{G}(y_j) = \sum_{\|y_j - c_k\| \leq r + r_x^k} \sum_{|\alpha| \leq p_{max} - 1} C_\alpha^k e^{-\|y_j - c_k\|^2/h^2} \left(\frac{y_j - c_k}{h}\right)^\alpha$;

Algorithm 2: The improved fast Gauss transform.

d	FGT					IFGT		
	# of boxes (N^d_{side})	p	# of terms (p^d)	n	Constant term	# of clusters (K)	p	# of terms ($r_{(p-1)d}$)
1	3	9	9	2	7.014806e+002	3	9	9
2	9	10	100	2	1.500000e+005	7	15	120
3	27	10	1000	2	1.948557e+007	15	16	816
4	81	11	14641	2	3.623648e+009	29	17	4845
5	243	11	161051	2	4.314985e+011	31	20	42504
6	729	12	2985984	2	9.069926e+013	62	20	177100
7	2187	14	105413504	2	3.774303e+016	67	22	1184040

Table 2.2: Comparison of the different parameters chosen by the FGT and the IFGT as a function of the data dimensionality d. $N = 100,000$ points were uniformly distributed in a unit hyper cube. The bandwidth was $h = 0.5$ and the target error was $\epsilon = 10^{-6}$.

	Small dimensions $d \leq 3$	Moderate dimensions $3 < d < 10$	Large dimensions $d \geq 10$
Small bandwidth $h \leq\approx 0.1$	FIGTree, Dual tree	FIGTree, Dual tree	FIGTree, Dual tree
Moderate bandwidth $0.1 \leq\approx h \leq\approx 0.5\sqrt{d}$	IFGT, FGT	IFGT	Direct
Large bandwidth $h \geq\approx 0.5\sqrt{d}$	IFGT, FGT	IFGT	IFGT

Table 2.3: Summary of the better performing algorithms for different settings of dimensionality d and bandwidth h (assuming data is scaled to a unit hypercube). The bandwidth ranges are approximate.

Chapter 3

Algorithm 2: Fast weighted summation of univariate Hermite $\times$ Gaussians

Most kernel methods require choosing some hyperparameters (e.g. bandwidth h of the kernel). Optimal procedures to choose these parameters scale as $\mathcal{O}(N^2)$. Most of these procedures involve solving some optimization which involves taking the derivatives of kernel sums. The derivatives of Gaussian sums involve sums of products of Hermite polynomials and Gaussians. In this chapter we describe a fast algorithm to compute the sums of products of univariate Hermite polynomials and Gaussians. [60, 61]

3.1 Introduction

Most state-of-the-art automatic bandwidth selection procedures for kernel density estimates and other hyperparameter selection procedures require the efficient computation of sums of Hermite times Gaussian. This sum arises when we differentiate the Gaussian.

Consider the following sum which we need to evaluate at M *target points*, $\{y_j \in \mathbf{R}\}_{j=1}^{M}$.

$$G_r(y_j) = \sum_{i=1}^{N} q_i H_r\left(\frac{y_j - x_i}{h_1}\right) e^{-(y_j-x_i)^2/h_2^2} \quad j = 1, \ldots, M, \tag{3.1}$$

where $\{q_i \in \mathbf{R}\}_{i=1}^{N}$ will be referred to as the *source weights*, $h_2 \in \mathbf{R}^+$ is the bandwidth of the Gaussian and $h_1 \in \mathbf{R}^+$ will be referred to as the bandwidth of the Hermite.

$H_r(u)$ is the rth Hermite polynomial. The Hermite polynomials are a set of orthogonal polynomials [1] . The first few Hermite polynomials are

$$H_0(u) \;=\; 1,\; H_1(u) = u, \text{ and } H_2(u) = u^2 - 1.$$

The computational complexity of evaluating Eq. 3.1 is $O(rNM)$. In this chapter we will present an $\epsilon - exact$ approximation algorithm that reduces the computational complexity to $O(prN + npr^2M)$, where the constants p and n depends on the precision ϵ and the bandwidth h. For any given $\epsilon > 0$ the algorithm computes an approximation $\widehat{G}_r(y_j)$ such that

$$\left|\frac{\hat{G}_r(y_j) - G_r(y_j)}{Q}\right| \leq \epsilon, \tag{3.2}$$

where $Q = \sum_{i=1}^{N} |q_i|$. We call $\widehat{G}_r(y_j)$ an $\epsilon - exact$ approximation to $G_r(y_j)$. The fast algorithm is based on separating the x_i and y_j in the Gaussian via the factorization

of the Gaussian by Taylor series and retaining only the first few terms so that the error due to truncation is less than the desired error. The Hermite function is factorized via the binomial theorem.

3.2 Factorization of the Gaussian

For any point $x_* \in \mathbf{R}$ the Gaussian can be written as,

$$\begin{aligned} e^{-\|y_j - x_i\|^2/h_2^2} &= e^{-\|(y_j - x_*)-(x_i - x_*)\|^2/h_2^2} \\ &= e^{-\|x_i - x_*\|^2/h_2^2} e^{-\|y_j - x_*\|^2/h_2^2} e^{2(x_i - x_*)(y_j - x_*)/h_2^2}. \end{aligned} \quad (3.3)$$

In Eq. 3.3 the first exponential $e^{-\|x_i - x_*\|^2/h^2}$ depends only on the source coordinates x_i. The second exponential $e^{-\|y_j - x_*\|^2/h^2}$ depends only on the target coordinates y_j. However for the third exponential $e^{2(y_j - x_*)(x_i - x_*)/h^2}$ the source and target are entangled. This entanglement is separated using the Taylor's series expansion.

The factorization of the Gaussian and the evaluation of the error bounds are based on the Taylor's series and Lagrange's evaluation of the remainder which we state here without the proof.

Theorem 4 [Taylor's Series] *For any point $x_* \in \mathbf{R}$, let $I \subset \mathbf{R}$ be an open set containing the point x_*. Let $f : I \to \mathbf{R}$ be a function which is n times differentiable on I. Then for any $x \in I$, there is a $\theta \in \mathbf{R}$ with $0 < \theta < 1$ such that*

$$f(x) = \sum_{k=0}^{n-1} \frac{1}{k!}(x - x_*)^k f^{(k)}(x_*) + \frac{1}{n!}(x - x_*)^n f^{(n)}(x_* + \theta(x - x_*)), \quad (3.4)$$

where $f^{(k)}$ is the kth derivative of the function f.

Based on the above theorem we have the following theorem.

Theorem 5 *Let $B_{r_x}(x_*)$ be a open interval of radius r_x with center $x_* \in \mathbf{R}$, i.e., $B_{r_x}(x_*) = \{x : \|x - x_*\| < r_x\}$. Let $h \in \mathbf{R}^+$ be a positive constant and $y \in \mathbf{R}$ be a fixed point such that $\|y - x_*\| < r_y$. For any $x \in B_{r_x}(x_*)$ and any non-negative integer p the function $f(x) = e^{2(x-x_*)(y-x_*)/h^2}$ can be written as*

$$f(x) \quad = \quad e^{2(x-x_*)(y-x_*)/h^2} = \sum_{k=0}^{p-1} \frac{2^k}{k!} \left(\frac{x - x_*}{h}\right)^k \left(\frac{y - x_*}{h}\right)^k + R_p(x), \tag{3.5}$$

and the residual

$$\begin{aligned} R_p(x) \quad &\leq \quad \frac{2^p}{p!} \left(\frac{\|x - x_*\|}{h}\right)^p \left(\frac{\|y - x_*\|}{h}\right)^p e^{2\|x-x_*\|\|y-x_*\|/h^2}. \\ &< \quad \frac{2^p}{p!} \left(\frac{r_x r_y}{h^2}\right)^p e^{2r_x r_y/h^2}. \end{aligned} \tag{3.6}$$

Proof: Let us define a new function $g(x) = e^{2[x(y-x_*)]/h^2}$. Using the result

$$g^{(k)}(x_*) = \frac{2^k}{h^k} e^{2[x_*(y-x_*)]/h^2} \left(\frac{y - x_*}{h}\right)^k \tag{3.7}$$

and Theorem 4, we have for any $x \in B_{r_x}(x_*)$ there is a $\theta \in \mathbf{R}$ with $0 < \theta < 1$ such that

$$\begin{aligned} g(x) \quad = \quad & e^{2[x_*(y-x_*)]/h^2} \left\{ \sum_{k=0}^{p-1} \frac{2^k}{k!} \left(\frac{x - x_*}{h}\right)^k \left(\frac{y - x_*}{h}\right)^k \right. \\ & \left. + \frac{2^p}{p!} \left(\frac{x - x_*}{h}\right)^p \left(\frac{y - x_*}{h}\right)^p e^{2\theta[(x-x_*).(y-x_*)]/h^2} \right\}. \end{aligned}$$

Hence

$$f(x) \quad = \quad e^{2(x-x_*)(y-x_*)/h^2} = \sum_{k=0}^{p-1} \frac{2^k}{k!} \left(\frac{x - x_*}{h}\right)^k \left(\frac{y - x_*}{h}\right)^k + R_p(x),$$

where,

$$R_p(x) = \frac{2^p}{p!} \left(\frac{x - x_*}{h}\right)^p \left(\frac{y - x_*}{h}\right)^p e^{2\theta[(x-x_*)(y-x_*)]/h^2}.$$

The remainder is bounded as follows.

$$\begin{aligned} R_p(x) &\leq \frac{2^p}{p!}\left(\frac{\|x-x_*\|}{h}\right)^p\left(\frac{\|y-x_*\|}{h}\right)^p e^{2\theta\|x-x_*\|\|y-x_*\|/h^2}, \\ &\leq \frac{2^p}{p!}\left(\frac{\|x-x_*\|}{h}\right)^p\left(\frac{\|y-x_*\|}{h}\right)^p e^{2\|x-x_*\|\|y-x_*\|/h^2} \text{ [Since } 0<\theta<1], \\ &< \frac{2^p}{p!}\left(\frac{r_x r_y}{h^2}\right)^p e^{2r_x r_y/h^2} \text{ [Since } \|x-x_*\|<r_x \text{ and } \|y-x_*\|<r_y]. \end{aligned}$$

Using Theorem 5 the Gaussian can now be factorized as

$$\begin{aligned} e^{-\|y_j-x_i\|^2/h_2^2} &= \sum_{k=0}^{p-1}\frac{2^k}{k!}\left[e^{-\|x_i-x_*\|^2/h_2^2}\left(\frac{x_i-x_*}{h_2}\right)^k\right]\left[e^{-\|y_j-x_*\|^2/h_2^2}\left(\frac{y_j-x_*}{h_2}\right)^k\right] \\ &+ \ error_p \end{aligned} \quad (3.8)$$

where,

$$error_p \leq \frac{2^p}{p!}\left(\frac{\|x_i-x_*\|}{h_2}\right)^p\left(\frac{\|y_j-x_*\|}{h_2}\right)^p e^{-(\|x_i-x_*\|-\|y_j-x_*\|)^2/h_2^2}. \quad (3.9)$$

3.3 Factorization of the Hermite polynomial

The rth Hermite polynomial can be written as [87]

$$H_r(x) = \sum_{l=0}^{\lfloor r/2\rfloor} a_l x^{r-2l}, \text{where } a_l = \frac{(-1)^l r!}{2^l l!(r-2l)!}.$$

Hence,

$$H_r\left(\frac{y_j-x_i}{h_1}\right) = \sum_{l=0}^{\lfloor r/2\rfloor} a_l\left(\frac{y_j-x_*}{h_1}-\frac{x_i-x_*}{h_1}\right)^{r-2l}.$$

Using the binomial theorem $(a+b)^n = \sum_{m=0}^{n}\binom{n}{m}a^m b^{n-m}$, the x_i and y_j can be separated as follows.

$$\left(\frac{y_j-x_*}{h_1}-\frac{x_i-x_*}{h_1}\right)^{r-2l} = \sum_{m=0}^{r-2l}(-1)^m\binom{r-2l}{m}\left(\frac{x_i-x_*}{h_1}\right)^m\left(\frac{y_j-x_*}{h_1}\right)^{r-2l-m}.$$

Substituting in the previous equation we have

$$H_r\left(\frac{y_j - x_i}{h_1}\right) = \sum_{l=0}^{\lfloor r/2 \rfloor}\sum_{m=0}^{r-2l} a_{lm}\left(\frac{x_i - x_*}{h_1}\right)^m \left(\frac{y_j - x_*}{h_1}\right)^{r-2l-m} \tag{3.10}$$

where,

$$a_{lm} = \frac{(-1)^{l+m} r!}{2^l l! m! (r-2l-m)!}. \tag{3.11}$$

3.4 Regrouping of the terms

Using Eq. 3.8 and 3.10, $G_r(y_j)$ after ignoring the error terms can be approximated as

$$\begin{aligned}\widehat{G}_r(y_j) &= \sum_{k=0}^{p-1}\sum_{l=0}^{\lfloor r/2 \rfloor}\sum_{m=0}^{r-2l} a_{lm}\left[\frac{2^k}{k!}\sum_{i=1}^{N} q_i e^{-\|x_i - x_*\|^2/h_2^2}\left(\frac{x_i - x_*}{h_2}\right)^k \left(\frac{x_i - x_*}{h_1}\right)^m\right] \\ &\quad \left[e^{-\|y_j - x_*\|^2/h_2^2}\left(\frac{y_j - x_*}{h_2}\right)^k \left(\frac{y_j - x_*}{h_1}\right)^{r-2l-m}\right] \\ &= \sum_{k=0}^{p-1}\sum_{l=0}^{\lfloor r/2 \rfloor}\sum_{m=0}^{r-2l} a_{lm} B_{km} e^{-\|y_j - x_*\|^2/h_2^2}\left(\frac{y_j - x_*}{h_2}\right)^k \left(\frac{y_j - x_*}{h_1}\right)^{r-2l-m}\end{aligned}$$

where

$$B_{km} = \frac{2^k}{k!}\sum_{i=1}^{N} q_i e^{-\|x_i - x_*\|^2/h_2^2}\left(\frac{x_i - x_*}{h_2}\right)^k \left(\frac{x_i - x_*}{h_1}\right)^m.$$

The coefficients B_{km} can be evaluated separately in $O(prN)$. Evaluation of $\widehat{G}_r(y_j)$ at M points is $O(pr^2M)$. Hence the computational complexity has reduced from the quadratic $O(rNM)$ to the linear $O(prN + pr^2M)$.

3.5 Space subdivision

Thus far, we have used the Taylor's series expansion about a certain point x_*. However if we use the same x_* for all the points we typically would require

very high truncation number p since the Taylor's series gives good approximation only in a small open interval around x_*. We uniformly sub-divide the space into K intervals of length $2r_x$. The N source points are assigned into K clusters, S_n for $n = 1, \ldots, K$ with c_n being the center of each cluster. The aggregated coefficients are now computed for each cluster and the total contribution from all the clusters is summed up.

$$\widehat{G}_r(y_j) = \sum_{n=1}^{K} \sum_{k=0}^{p-1} \sum_{l=0}^{\lfloor r/2 \rfloor} \sum_{m=0}^{r-2l} a_{lm} B_{km}^{n} e^{-\|y_j - c_n\|^2/h_2^2} \left(\frac{y_j - c_n}{h_2} \right)^k \left(\frac{y_j - c_n}{h_1} \right)^{r-2l-m} \quad (3.12)$$

where,

$$B_{km}^{n} = \frac{2^k}{k!} \sum_{x_i \in S_n} q_i e^{-\|x_i - c_n\|^2/h_2^2} \left(\frac{x_i - c_n}{h_2} \right)^k \left(\frac{x_i - c_n}{h_1} \right)^m . \quad (3.13)$$

3.6 Decay of the Gaussian

Since the Gaussian decays very rapidly a further speedup is achieved if we ignore all the sources belonging to a cluster if the cluster is greater than a certain distance from the target point, i.e., $\|y_j - c_n\| > r_y$. The cluster cutoff radius r_y depends on the desired error ϵ. Substituting $h_1 = h$ and $h_2 = \sqrt{2}h$ we have

$$\widehat{G}_r(y_j) = \sum_{\|y_j - c_n\| \leq r_y} \sum_{k=0}^{p-1} \sum_{l=0}^{\lfloor r/2 \rfloor} \sum_{m=0}^{r-2l} a_{lm} B_{km}^{n} e^{-\|y_j - c_n\|^2/2h^2} \left(\frac{y_j - c_n}{h} \right)^{k+r-2l-m} \quad (3.14)$$

where,

$$B_{km}^{n} = \frac{1}{k!} \sum_{x_i \in S_n} q_i e^{-\|x_i - c_n\|^2/2h^2} \left(\frac{x_i - c_n}{h} \right)^{k+m} \quad (3.15)$$

and

$$a_{lm} = \frac{(-1)^{l+m} r!}{2^l l! m! (r - 2l - m)!} . \quad (3.16)$$

3.7 Computational and space complexity

Computing the coefficients B^n_{km} for all the clusters is $O(prN)$. Evaluation of $\widehat{G}_r(y_j)$ at M points is $O(npr^2M)$, where n if the maximum number of neighbor clusters which influence y_j. Hence the total computational complexity is $O(prN + npr^2M)$. Assuming $N = M$ the total computational complexity is $O(cN)$ where the constant $c = pr + npr^2$ depends on the desired error, the bandwidth, and r. For each cluster we need to store all the pr coefficients. Hence the storage needed is of $O(prK + N + M)$.

3.8 Error bounds and choosing the parameters

Given any $\epsilon > 0$, we want to choose the following parameters, r_x (the interval length), r_y (the cut off radius for each cluster), and p (the truncation number) such that for any target point y_j

$$\left| \frac{\hat{G}_r(y_j) - G_r(y_j)}{Q} \right| \leq \epsilon, \tag{3.17}$$

where $Q = \sum_{i=1}^{N} |q_i|$. Let us define Δ_{ij} to be the point wise error in $\widehat{G}_r(y_j)$ contributed by the ith source x_i. We now require that

$$|\hat{G}_r(y_j) - G_r(y_j)| = \left| \sum_{i=1}^{N} \Delta_{ij} \right| \leq \sum_{i=1}^{N} |\Delta_{ij}| \leq \sum_{i=1}^{N} |q_i|\epsilon. \tag{3.18}$$

One way to achieve this is to let

$$|\Delta_{ij}| \leq |q_i|\epsilon \;\; \forall i = 1, \ldots, N.$$

Let c_n be the center of the cluster to which x_i belongs. There are two different ways in which a source can contribute to the error. The first is due to ignoring the cluster

S_n if it is outside a given radius r_y from the target point y_j. In this case,

$$\Delta_{ij} = q_i H_r\left(\frac{y_j - x_i}{h}\right) e^{-\|y_j - x_i\|^2/2h^2}. \tag{3.19}$$

For all clusters which are within a distance r_y from the target point the error is due to the truncation of the Taylor's series after order $p-1$. From Eq. 3.9 and using the fact that $h_1 = h$ and $h_2 = \sqrt{2}h$ we have,

$$\Delta_{ij} \leq \frac{q_i}{p!} H_r\left(\frac{y_j - x_i}{h}\right)\left(\frac{\|x_i - c_n\|}{h}\right)^p\left(\frac{\|y_j - c_n\|}{h}\right)^p e^{-(\|x_i - c_n\| - \|y_j - c_n\|)^2/2h^2}. \tag{3.20}$$

3.8.1 Choosing the cut-off radius

We want to choose the cut-off radius such that (Eq. 3.19)

$$\left|H_r\left(\frac{y_j - x_i}{h}\right)\right| e^{-\|y_j - x_i\|^2/2h^2} \leq \epsilon \tag{3.21}$$

We use the following inequality to bound the Hermite polynomial [3].

$$\left|H_r\left(\frac{y_j - x_i}{h}\right)\right| \leq \sqrt{r!} e^{\|y_j - x_i\|^2/4h^2}. \tag{3.22}$$

Substituting this bound in Eq. 7.7 we have

$$e^{-\|y_j - x_i\|^2/4h^2} \leq \epsilon/\sqrt{r!}. \tag{3.23}$$

This implies that $\|y_j - x_i\| > 2h\sqrt{\ln(\sqrt{r!}/\epsilon)}$. Using the reverse triangle inequality, $\|a-b\| \geq \big|\|a\| - \|b\|\big|$, and the fact that $\|y_j - c_n\| > r_y$ and $\|x_i - c_n\| \leq r_x$, we have

$$\|y_j - x_i\| = \|(y_j - c_n) - (x_i - c_n)\| \geq \big|\|(y_j - c_n)\| - \|(x_i - c_n)\|\big| > \big|r_y - r_x\big| \tag{3.24}$$

So in order that the error due to ignoring the faraway clusters is less than $|q_i|\epsilon$ we have to choose r_y such that

$$\left|r_y - r_x\right| > 2h\sqrt{\ln\left(\sqrt{r!}/\epsilon\right)}. \tag{3.25}$$

If we choose $r_y > r_x$ then,

$$r_y > r_x + 2h\sqrt{\ln\left(\sqrt{r!}/\epsilon\right)}. \tag{3.26}$$

3.8.2 Choosing the truncation number

For a given source x_i we have to choose the truncation number p such that $|\Delta_{ij}| \leq |q_i|\epsilon$. Δ_{ij} depends both on distance between the source and the cluster center, i.e., $\|x_i - c_n\|$ and the distance between the target and the cluster center, i.e., $\|y_j - c_n\|$.

For all sources for which $\|y_j - c_n\| \leq r_y$ we have

$$\Delta_{ij} \leq \frac{q_i}{p!} H_r\left(\frac{y_j - x_i}{h}\right)\left(\frac{\|x_i - c_n\|}{h}\right)^p\left(\frac{\|y_j - c_n\|}{h}\right)^p e^{-(\|x_i - c_n\| - \|y_j - c_n\|)^2/2h^2}. \tag{3.27}$$

Using the bound on the Hermite polynomial (Eq. 3.22) this can be written as

$$|\Delta_{ij}| \leq \frac{|q_i|\sqrt{r!}}{p!}\left(\frac{\|x_i - c_n\|}{h}\right)^p\left(\frac{\|y_j - c_n\|}{h}\right)^p e^{-(\|x_i - c_n\| - \|y_j - c_n\|)^2/4h^2}. \tag{3.28}$$

The speedup is achieved because at each cluster S_n we sum up the effect of all the sources. As a result we do not have a knowledge of $\|y_j - c_n\|$. So we will have to bound the right hand side of Eq. 3.28, such that it is independent of $\|y_j - c_n\|$.

Fig. 3.1 shows the error at y_j due to source x_i, i.e., $|\Delta_{ij}|$ [Eq. 3.28] as a function of $\|y_j - c_n\|$ for different values of p and for $h = 0.1$ and $r = 4$. The error increases as a function of $\|y_j - c_n\|$, reaches a maximum and then starts decreasing. The maximum is attained at (obtained by taking the first derivative of the R.H.S. of Eq. 3.28 and setting it to zero),

$$\|y_j - c_n\|_* = \frac{\|x_i - c_n\| + \sqrt{\|x_i - c_n\|^2 + 8ph^2}}{2} \tag{3.29}$$

Hence we choose p such that,

$$|\Delta_{ij}| \left|_{[\|y_j - c_n\| = \|y_j - c_n\|_*]} \leq |q_i|\epsilon. \right. \tag{3.30}$$

In case $\|y_j - c_n\|_* > r_y$ we need to choose p based on r_y, since Δ_{ij} will be much lower there. Hence out strategy for choosing p is ,

$$|\Delta_{ij}| \left|_{[\|y_j - c_n\| = \min(\|y_j - c_n\|_*, r_y),\, \|x_i - c_n\| = r_x]} \leq |q_i|\epsilon. \right. \tag{3.31}$$

3.8.3 Choosing the interval length

The only free parameter remaining the interval length r_x. We set $r_x = h/2$. If the truncation number chosen is very high then r_x can be reduced. The final algorithm is summarized below.

1. Scale the N data points $\{x_i\}_{i=1}^N$ to lie in the unit interval $[0, 1]$.

2. Choose $r_x = h/2$. Sub-divide the unit interval into K intervals of length $2r_x$. The N source points are assigned into K clusters, S_n for $n = 1, \ldots, K$ with c_n being the center of each cluster.

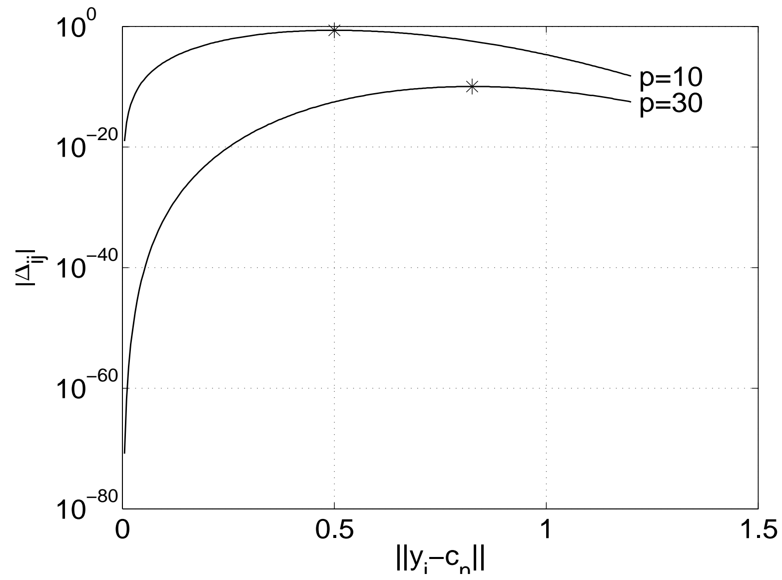


Figure 3.1: The error at y_j due to source x_i, i.e., Δ_{ij} [Eq. 3.28] as a function of $\|y_j - c_n\|$ for different values of p and for $h = 0.1$ and $r = 4$. The error increases as a function of $\|y_j - c_n\|$, reaches a maximum and then starts decreasing. The maximum is marked as '*'. $q_i = 1$ and $\|x_i - c_n\| = 0.1$.

3. Choose the cutoff radius $r_y = r_x + 2h\sqrt{\ln(\sqrt{r!}/\epsilon)}$.

4. Choose the truncation number p such that

$$\frac{\sqrt{r!}}{p!}\left(\frac{r_x b}{h^2}\right)^p e^{-(r_x-b)^2/4h^2} \leq \epsilon, \text{ where} b = min\left(r_y, \frac{r_x + \sqrt{r_x^2 + 8ph^2}}{2}\right). \quad (3.32)$$

5. For each cluster S_n compute the aggregated coefficients B_{km}^n for $k = 0, \ldots, p-1$ and $m = 0, \ldots, r$.

$$B_{km}^n = \frac{1}{k!}\sum_{x_i \in S_n} q_i e^{-\|x_i - c_n\|^2/2h^2}\left(\frac{x_i - c_n}{h}\right)^{k+m} \quad (3.33)$$

6. Compute the coefficients a_{lm} for $l = 0, \ldots, \lfloor r/2 \rfloor$ and $m = 0, \ldots, r$.

$$a_{lm} = \frac{(-1)^{l+m} r!}{2^l l! m! (r - 2l - m)!}. \quad (3.34)$$

7. The approximate the kernel density derivative at point y_j is computed as

$$\widehat{G}_r(y_j) = \sum_{\|y_j - c_n\| \leq r_y} \sum_{k=0}^{p-1} \sum_{l=0}^{\lfloor r/2 \rfloor} \sum_{m=0}^{r-2l} a_{lm} B_{km}^n e^{-\|y_j - c_n\|^2/2h^2}\left(\frac{y_j - c_n}{h}\right)^{k+r-2l-m} \quad (3.35)$$

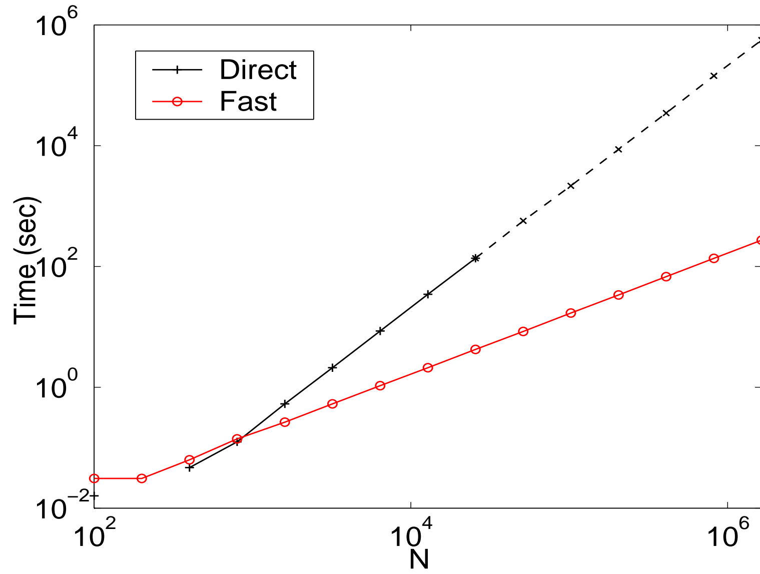
3.9 Numerical experiments

In this section we present some numerical studies of the speedup and the actual error as a function of the number of data points, the bandwidth h, the order r, and the desired error ϵ. The algorithm was programmed in C++ with MATLAB bindings and was run on 2.4 GHz processor with 2 GB of RAM. The code is available under LGPL at `http://www.umiacs.umd.edu/~vikas/Software/optimal_bw/optimal_bw_code.htm`.

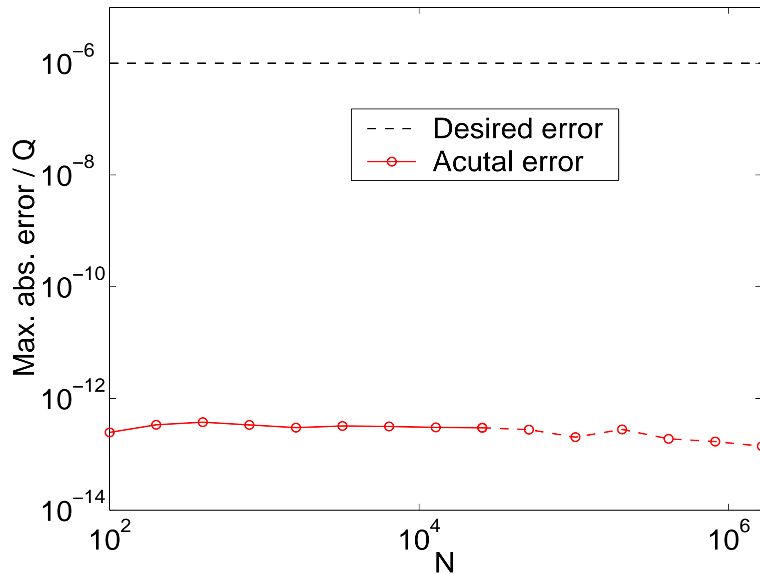
Fig. 4.4 shows the running time and the maximum absolute error relative to Q for both the direct and the fast methods as a function of $N = M$. The bandwidth

was $h = 0.1$ and the order of the derivative was $r = 4$. The source and the target points were uniformly distributed in the unit interval. We see that the running time of the fast method grows linearly as the number of sources and targets increases, while that of the direct evaluation grows quadratically. We also observe that the error is way below the desired error thus validating our bound.

Fig. 4.5 shows the tradeoff between precision and speedup. An increase in speedup is obtained at the cost of reduced accuracy. Fig. 3.4 shows the results as a function of bandwidth h. Better speedup is obtained at larger bandwidths. Fig. 3.5 shows the results for different orders of the density derivatives.

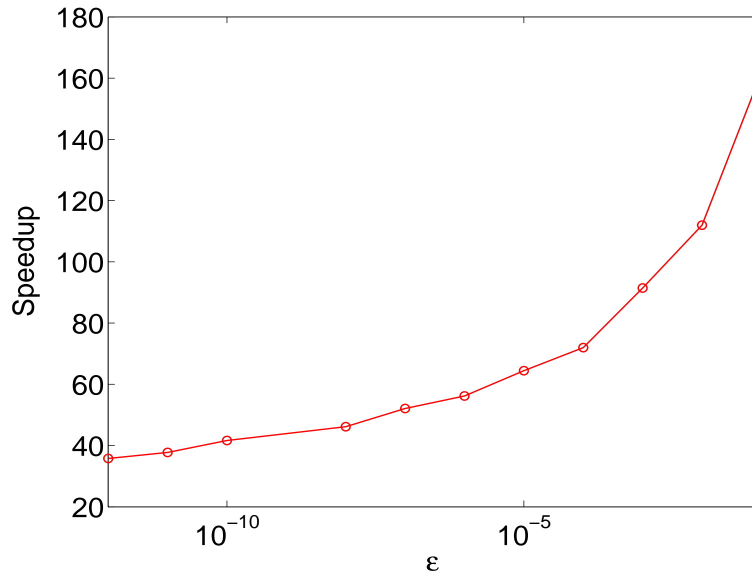


(a)

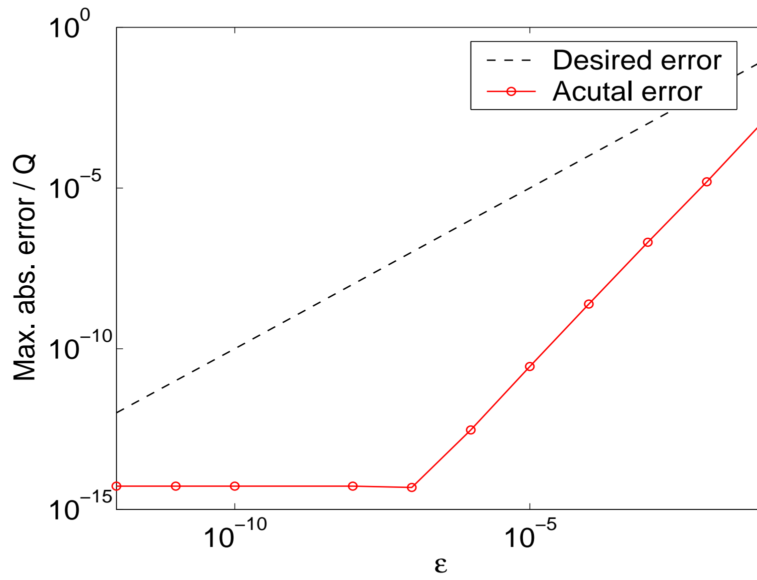


(b)

Figure 3.2: (a) The running time in seconds and (b) maximum absolute error relative to Q for the direct and the fast methods as a function of N. $N = M$ source and the target points were uniformly distributed in the unit interval. For $N > 25,600$ the timing results for the direct evaluation were obtained by evaluating the result at $M = 100$ points and then extrapolating. [$h = 0.1$, $r = 4$, and $\epsilon = 10^{-6}$.]

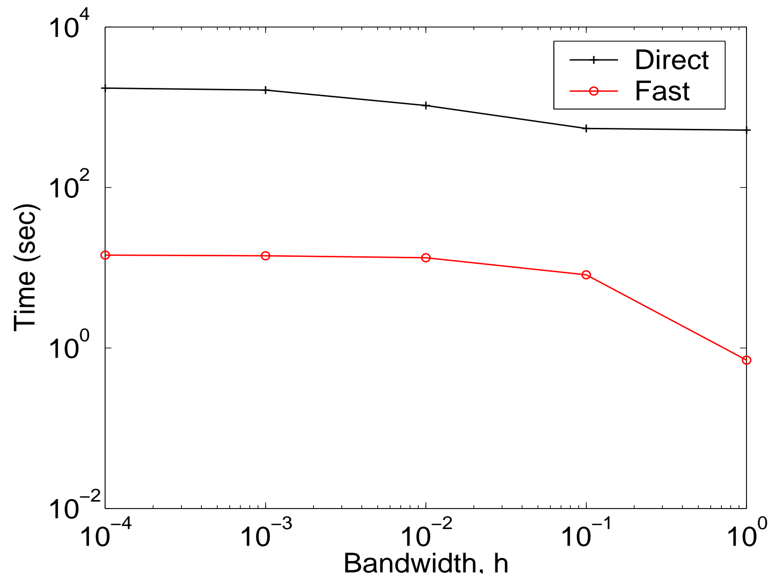


(a)

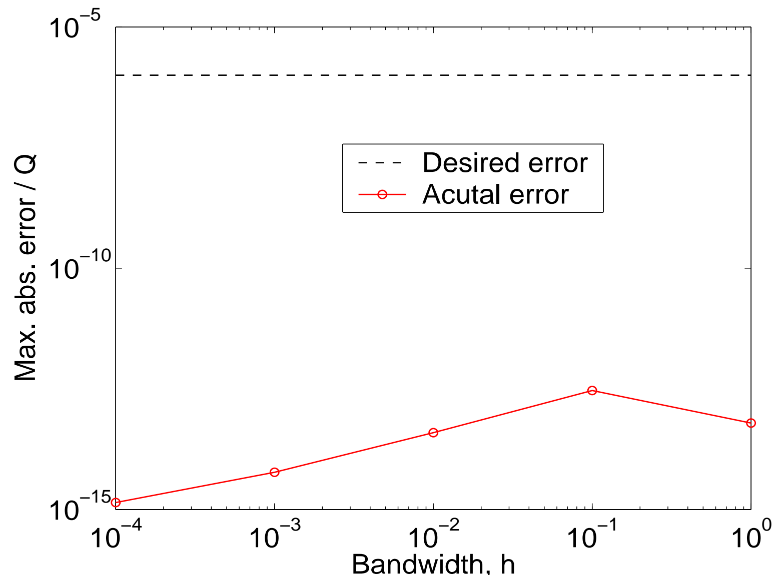


(b)

Figure 3.3: (a) The speedup achieved and (b) maximum absolute error relative to Q for the direct and the fast methods as a function of ϵ. $N = M = 50,000$ source and the target points were uniformly distributed in the unit interval. [$h = 0.1$ and $r = 4$]

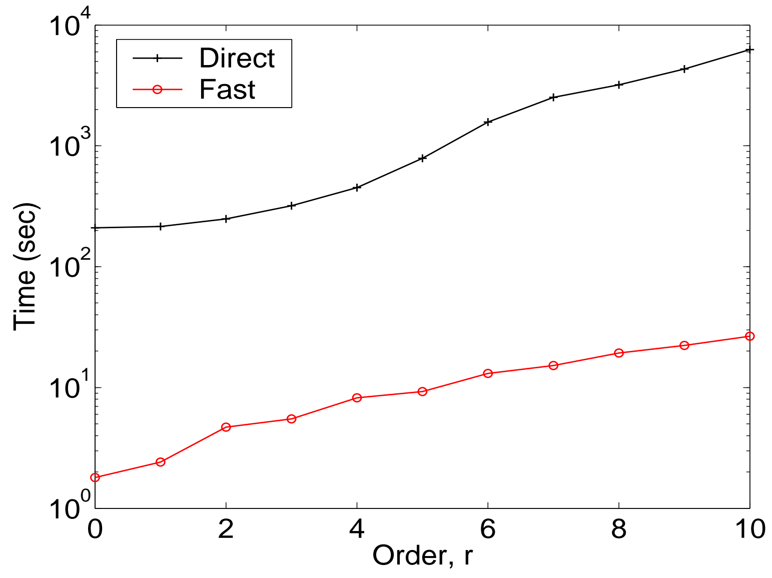


(a)



(b)

Figure 3.4: (a) The running time in seconds and (b) maximum absolute error relative to Q for the direct and the fast methods as a function of h. $N = M = 50,000$ source and the target points were uniformly distributed in the unit interval. [$\epsilon = 10^{-6}$ and $r = 4$]



(a)

(b)

Figure 3.5: (a) The running time in seconds and (b) maximum absolute error relative to Q for the direct and the fast methods as a function of r. $N = M = 50,000$ source and the target points were uniformly distributed in the unit interval. [$\epsilon = 10^{-6}$ and $h = 0.1$]

Chapter 4

Algorithm 3: Fast weighted summation of erfc functions

Direct computation of the weighted sum of N complementary error functions at M points scales as $\mathcal{O}(MN)$. We present a $\mathcal{O}(M+N)$ ϵ-exact approximation algorithm to compute the same [63]. We have encountered this sum in a ranking problem.

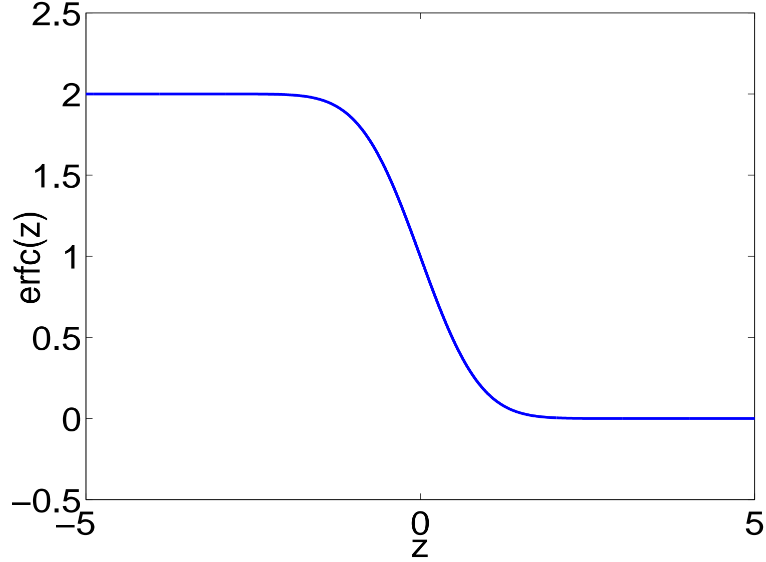


Figure 4.1: The erfc function.

4.1 Introduction

The complementary error function is defined as follows (see Figure 4.1) [1]

$$\operatorname{erfc}(z) = \frac{2}{\sqrt{\pi}} \int_z^\infty e^{-t^2} dt. \tag{4.1}$$

Consider the following weighted summation of N erfc functions each centered at $\{x_i\}_{i=1}^N$.

$$E(y) = \sum_{i=1}^{N} q_i \operatorname{erfc}(y - x_i). \tag{4.2}$$

The scalars q_i will be referred to as the *weights*. Direct computation of (8.18) at M points $\{y_j\}_{j=1}^M$ is $\mathcal{O}(MN)$. In this report we will derive an ϵ-exact approximation algorithm to compute the same in $\mathcal{O}(M+N)$ time.

For any given $\epsilon > 0$, $\widehat{E}$ is an $\epsilon - exact$ approximation to E if the maximum absolute error relative to the total weight $Q_{abs} = \sum_{i=1}^N |q_i|$ is upper bounded by ϵ,

i.e.,

$$\max_{y_j} \left[\frac{|\hat{E}(y_j) - E(y_j)|}{Q_{abs}} \right] \le \epsilon. \tag{4.3}$$

The constant in $\mathcal{O}(M+N)$, depends on the desired *accuracy* ϵ, which however can be *arbitrary*. In fact for machine precision accuracy there is no difference between the direct and the fast methods. The algorithm is inspired by the *fast multipole methods* proposed in computational physics [27]. The fast algorithm is based on using a infinite series expansion for the erfc function and retaining only the first few terms contributing to the desired accuracy ϵ.

4.2 Series expansion

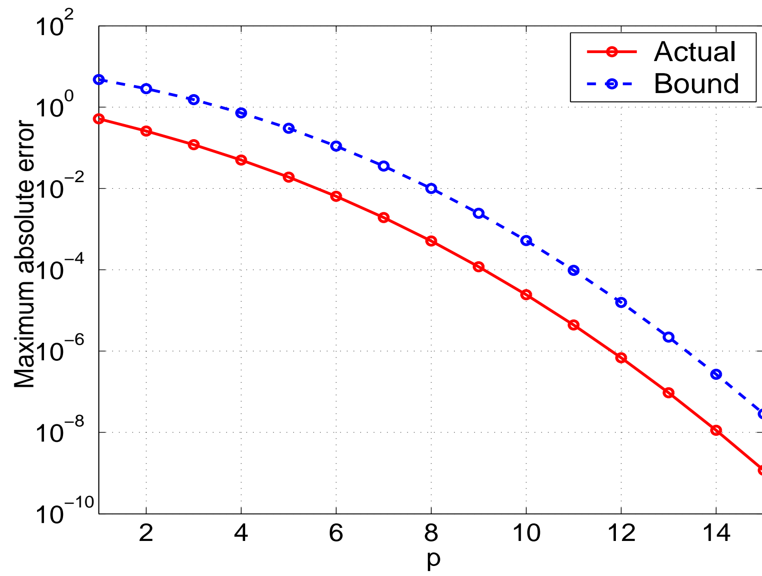
Several series exist for the erfc function (See for e.g. Chapter 7 in [1]). Some are applicable only to a restricted interval, while other need a large number of terms to converge. We use the following series derived by Beauliu [4, 80].

$$\mathrm{erfc}(x) = 1 - \frac{4}{\pi} \sum_{\substack{n=1 \\ n \text{ odd}}}^{2p-1} \frac{e^{-n^2h^2}}{n} \sin(2nhx) + \mathrm{error}(x), \tag{4.4}$$

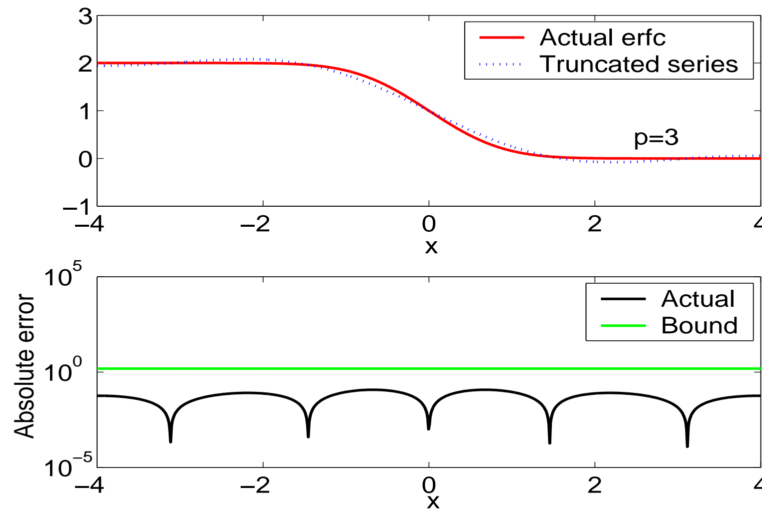
where

$$|\mathrm{error}(x)| < \left| \frac{4}{\pi} \sum_{\substack{n=2p+1 \\ n \text{ odd}}}^{\infty} \frac{e^{-n^2h^2}}{n} \sin(2nhx) \right| + \mathrm{erfc}\left(\frac{\pi}{2h} - |x| \right). \tag{4.5}$$

Here, p is the *truncation number* and h is a real number related to the sampling interval. These kind of series are of interest in the field digital communications wherein the noise is modeled as a Gaussian random variable. The series is derived by applying a Chernoff bound approach to an approximate Fourier series expansion of a periodic square waveform [4].



(a)



(b)

Figure 4.2: (a) The maximum absolute error between the actual value of erfc and the truncated series representation (Eq. 7.6) as a function of the truncation number p for any $x \in [-4, 4]$. The error bound (Eq. 4.6) is also shown. (b) A sample plot of the actual erfc function and the $p = 3$ truncated series representation. The error as a function of x is also shown in the lower panel.

This series converges rapidly especially as $x \to 0$. Figure 4.2(a) shows the maximum absolute error between the actual value of erfc [1] and the truncated series representation as a function of p. For example for any $x \in [-4, 4]$ with $p = 12$ the error is less than 10^{-6}. We have to choose p and h such that the error has to be less than ϵ. We further bound the first term in (4.5) as follows.

$$
\begin{aligned}
\left| \frac{4}{\pi} \sum_{\substack{n=2p+1 \\ n \text{ odd}}}^{\infty} \frac{e^{-n^2h^2}}{n} \sin(2nhx) \right| &\leq \frac{4}{\pi} \sum_{\substack{n=2p+1 \\ n \text{ odd}}}^{\infty} \frac{e^{-n^2h^2}}{n} |\sin(2nhx)| \\
&\leq \frac{4}{\pi} \sum_{\substack{n=2p+1 \\ n \text{ odd}}}^{\infty} \frac{e^{-n^2h^2}}{n} \quad [\text{ Since } |\sin(2nhx)| \leq 1] \\
&< \frac{4}{\pi} \sum_{\substack{n=2p+1 \\ n \text{ odd}}}^{\infty} e^{-n^2h^2} \\
&< \frac{4}{\pi} \int_{2p+1}^{\infty} e^{-x^2h^2} dx < \frac{2}{\sqrt{\pi}h} \left[\frac{2}{\sqrt{\pi}} \int_{(2p+1)h}^{\infty} e^{-t^2} dt \right] \\
&= \frac{2}{\sqrt{\pi}h} \text{erfc}((2p+1)h).
\end{aligned}
$$

Hence

$$
|\text{error}(x)| < \frac{2}{\sqrt{\pi}h} \text{erfc}\left((2p+1)h\right) + \text{erfc}\left(\frac{\pi}{2h} - |x|\right). \tag{4.6}
$$

For a fixed p and h as $|x|$ increases the error increases. Therefore as $|x|$ increases, h should decrease and consequently the series converges slower leading to a large truncation number p.

[1]There is no closed form expression to compute erfc directly. The implementation in MATLAB uses a rational Chebyshev approximation and the accuracy is not adequate enough. In order to compare the error between the actual and the series approximation we use the Maple implementation(`feval(maple('erfc'),x))`) which provides very high precision using symbolic integration.

4.3 Fast summation algorithm

We will now derive a fast algorithm to compute $E(y)$ based on the series (7.6).

$$\begin{aligned} E(y) &= \sum_{i=1}^{N} q_i \mathrm{erfc}(y - x_i) \\ &= \sum_{i=1}^{N} q_i \left[1 - \frac{4}{\pi} \sum_{\substack{n=1 \\ n \text{ odd}}}^{2p-1} \frac{e^{-n^2h^2}}{n} \sin\{2nh(y - x_i)\} + \text{error} \right]. \end{aligned} \tag{4.7}$$

Ignoring the error term the sum $E(y)$ can be approximated as

$$\widehat{E}(y) = Q - \frac{4}{\pi} \sum_{i=1}^{N} q_i \sum_{\substack{n=1 \\ n \text{ odd}}}^{2p-1} \frac{e^{-n^2h^2}}{n} \sin\{2nh(y - x_i)\}, \tag{4.8}$$

where $Q = \sum_{i=1}^{N} q_i$. The terms y and x_i appear together in the argument of the sin function. We separate them using the trigonometric identity $\sin(\alpha - \beta) = \sin(\alpha)\cos(\beta) - \cos(\alpha)\sin(\beta)$.

$$\begin{aligned} \sin\{2nh(y - x_i)\} &= \sin\{2nh(y - x_*) - 2nh(x_i - x_*)\} \\ &= \sin\{2nh(y - x_*)\}\cos\{2nh(x_i - x_*)\} \\ &\quad - \cos\{2nh(y - x_*)\}\sin\{2nh(x_i - x_*)\}. \end{aligned} \tag{4.9}$$

Note that we have shifted all the points by x_*. Substituting the separated representation (4.9) in Eq. 4.8 we have

$$\begin{aligned} \widehat{E}(y) &= Q - \frac{4}{\pi} \sum_{i=1}^{N} q_i \sum_{\substack{n=1 \\ n \text{ odd}}}^{2p-1} \frac{e^{-n^2h^2}}{n} \sin\{2nh(y - x_*)\}\cos\{2nh(x_i - x_*)\} \\ &\quad + \frac{4}{\pi} \sum_{i=1}^{N} q_i \sum_{\substack{n=1 \\ n \text{ odd}}}^{2p-1} \frac{e^{-n^2h^2}}{n} \cos\{2nh(y - x_*)\}\sin\{2nh(x_i - x_*)\}. \end{aligned} \tag{4.10}$$

Exchanging the order of summation and regrouping the terms we have the following expression.

$$\widehat{E}(y) = Q - \frac{4}{\pi} \sum_{\substack{n=1 \\ n \text{ odd}}}^{2p-1} \left[A_n \sin\{2nh(y - x_*)\} - B_n \cos\{2nh(y - x_*)\} \right]. \quad (4.11)$$

where

$$A_n = \frac{e^{-n^2h^2}}{n} \sum_{i=1}^{N} q_i \cos\{2nh(x_i - x_*)\} \text{ and } B_n = \frac{e^{-n^2h^2}}{n} \sum_{i=1}^{N} q_i \sin\{2nh(x_i - x_*)\} \quad (4.12)$$

4.4 Runtime and storage analysis

Note that the coefficients $\{A_n, B_n\}_{n=1(n \text{ odd})}^{2p-1}$ do not depend on y. Hence each of A_n and B_n can be evaluated separately is $\mathcal{O}(N)$ time. Since there are p such coefficients the total complexity to compute A and B is $\mathcal{O}(2pN)$. The term $Q = \sum_{i=1}^{N} q_i$ can also be precomputed in $\mathcal{O}(N)$ time. Once A, B, and Q have been precomputed, evaluation of $\widehat{E}(y)$ requires $\mathcal{O}(2p)$ operations. Evaluating at M points is $\mathcal{O}(2pM)$. Hence the computational complexity has reduced from the quadratic $\mathcal{O}(NM)$ to the linear $\mathcal{O}((2p+1)N + 2pM)$. We need space to store the points and the coefficients A and B. Hence the storage complexity is $\mathcal{O}(N + M + 2p)$.

4.5 Direct inclusion and exclusion of faraway points

Note that $z = (y - x_i) \in [-\infty, \infty]$. The truncation number p required to approximate $\text{erfc}(z)$ can be quite large for large $|z|$. Luckily $\text{erfc}(z) \to 2$ as $z \to -\infty$ and $\text{erfc}(z) \to 0$ as $z \to \infty$ very quickly [See Figure 4.3(a)]. Since we are interested

in the result only to a certain precision ϵ we can use the following approximation.

$$\operatorname{erfc}(z) \approx \begin{cases} 2 & \text{if } z < -r \\ p\text{-truncated series} & \text{if } -r \leq z \leq r \\ 0 & \text{if } z > r \end{cases} \tag{4.13}$$

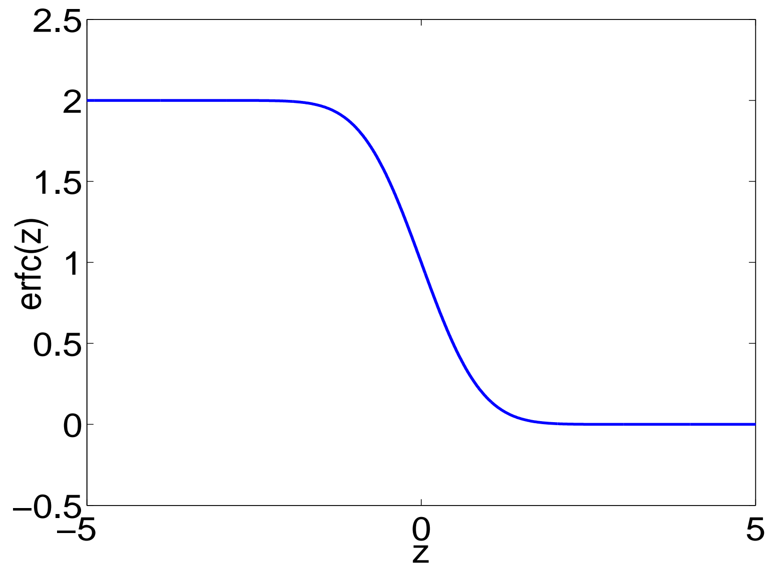
The bound r and the truncation number p have to be chosen such that for any z the error is always less than ϵ. From Figure 4.3(b) we can see that for error of the order 10^{-15} we need to use the series expansion for $-6 \leq z \leq 6$.

However we cannot check the value of $(y - x_i)$ for all pairs of x_i and y. This would lead us back to the quadratic complexity. To avoid this, we subdivide the points into clusters.

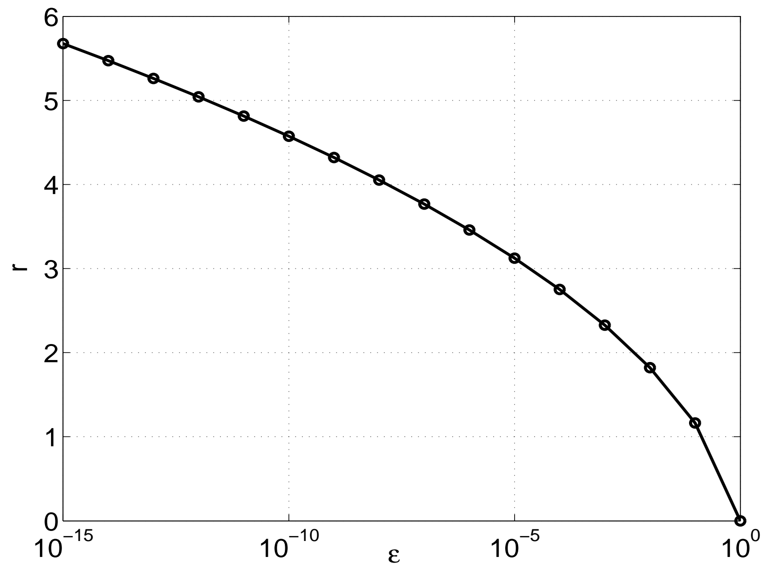
4.6 Space sub-division

We uniformly sub-divide the space into K intervals of length $2r_x$. The N source points are assigned into K clusters, S_k for $k = 1, \ldots, K$ with c_k being the center of each cluster. The aggregated coefficients are now computed for each cluster and the total contribution from all the influential clusters is summed up. For each cluster if $|y - c_k| \leq r_y$ then we will incorporate the series coefficients. If $(y - c_k) < -r_y$ then we will include a contribution of $2Q_k$. If $(y - c_k) > r_y$ then we will ignore that cluster. Hence

$$\begin{aligned} \widehat{E}(y) &= \sum_{|y-c_k| \leq r_y} \left(Q_k - \frac{4}{\pi} \sum_{\substack{n=1 \\ n \text{ odd}}}^{2p-1} \left[A_n^k \sin\{2nh(y - c_k)\} - B_n^k \cos\{2nh(y - c_k)\} \right] \right) \\ &+ \sum_{(y-c_k) < -r_y} 2Q_k. \end{aligned} \tag{4.14}$$



(a)



(b)

Figure 4.3: (a) The erfc function (b) The value of r for which $\text{erfc}(z) < \epsilon, \ \forall z > r$

where

$$
\begin{aligned}
A_n^k &= \frac{e^{-n^2h^2}}{n}\sum_{i=1}^{N} q_i \cos\{2nh(x_i - c_k)\}, \\
B_n^k &= \frac{e^{-n^2h^2}}{n}\sum_{i=1}^{N} q_i \sin\{2nh(x_i - c_k)\}, \text{ and} \\
Q_k &= \sum_{\forall x_i \in S_k} q_i. \qquad (4.15)
\end{aligned}
$$

The computational complexity to compute A,B, and Q is still $\mathcal{O}((2p+1)N)$ since each x_i belongs to only one cluster. Let l be the number of influential clusters, *i.e.*, the clusters for which $|y - c_k| \leq r_y$. Evaluating $\widehat{E}(y)$ at M points due to these l clusters is $\mathcal{O}(2plM)$. Let m be the number of clusters for which $(y - c_k) < -r_y$. Evaluating $\widehat{E}(y)$ at M points due to these m clusters is $\mathcal{O}(mM)$. Hence the total computational complexity is $\mathcal{O}((2p+1)N + (2pl+m)M)$. The storage complexity is $\mathcal{O}(N + M + (2p+1)K)$.

4.7 Choosing the parameters

Given any $\epsilon > 0$, we want to choose the following parameters,

- r_x (the interval length),
- r (the cut off radius),
- p (the truncation number), and
- h such that

for *any* target point y

$$\left| \frac{\hat{E}(y) - E(y)}{Q_{abs}} \right| \leq \epsilon, \qquad (4.16)$$

where $Q_{abs} = \sum_{i=1}^{N} |q_i|$.

Let us define Δ_i to be the point wise error in $\widehat{E}(y)$ contributed by the ith source x_i. We now require that

$$|\hat{E}(y) - E(y)| = \left| \sum_{i=1}^{N} \Delta_i \right| \leq \sum_{i=1}^{N} |\Delta_i| \leq \sum_{i=1}^{N} |q_i|\epsilon. \tag{4.17}$$

One way to achieve this is to let $|\Delta_i| \leq |q_i|\epsilon \ \forall i = 1, \ldots, N$.

For all x_i such that $|y - x_i| \leq r$ we have

$$|\Delta_i| < \underbrace{|q_i| \frac{2}{\sqrt{\pi} h} \operatorname{erfc}\left((2p+1)h\right)}_{T_e} + \underbrace{|q_i| \operatorname{erfc}\left(\frac{\pi}{2h} - r\right)}_{S_e}. \tag{4.18}$$

We have to choose the parameters such that $|\Delta_i| < |q_i|\epsilon$. We will let $S_e < |q_i|\epsilon/2$. This implies that

$$\frac{\pi}{2h} - r > \operatorname{erfc}^{-1}(\epsilon/2). \tag{4.19}$$

Hence we have to choose

$$h < \frac{\pi}{2\left(r + \operatorname{erfc}^{-1}(\epsilon/2)\right)}. \tag{4.20}$$

We will choose

$$h = \frac{\pi}{3\left(r + \operatorname{erfc}^{-1}(\epsilon/2)\right)}. \tag{4.21}$$

We will choose p such that $T_e < |q_i|\epsilon/2$. This implies that

$$2p + 1 > \frac{1}{h} \operatorname{erfc}^{-1}\left(\frac{\sqrt{\pi} h \epsilon}{4}\right). \tag{4.22}$$

We choose

$$p = \left\lceil \frac{1}{2h} \operatorname{erfc}^{-1}\left(\frac{\sqrt{\pi} h \epsilon}{4}\right) \right\rceil. \tag{4.23}$$

Note that as r increases h decreases and consequently p increases. If $x \in (r, \infty]$ we approximate $\operatorname{erfc}(x)$ by 0 and if $x \in [-\infty, -r)$ then approximate $\operatorname{erfc}(x)$ by 2. If we

choose

$$r > \operatorname{erfc}^{-1}(\epsilon), \tag{4.24}$$

then the approximation will result in a error $< \epsilon$. In practice we choose

$$r = \operatorname{erfc}^{-1}(\epsilon) + 2r_x, \tag{4.25}$$

where r_x is the cluster radius. For a target point y the number of influential clusters

$$(2l+1) = \left\lceil \frac{2r}{2r_x} \right\rceil. \tag{4.26}$$

Let us choose $r_x = 0.1\operatorname{erfc}^{-1}(\epsilon)$. This implies $2l+1 = 12$. So we have to consider $n = 6$ clusters on either side of the target point. Summarizing the parameters are given by

1. $r_x = 0.1\operatorname{erfc}^{-1}(\epsilon)$.

2. $r = \operatorname{erfc}^{-1}(\epsilon) + 2r_x$.

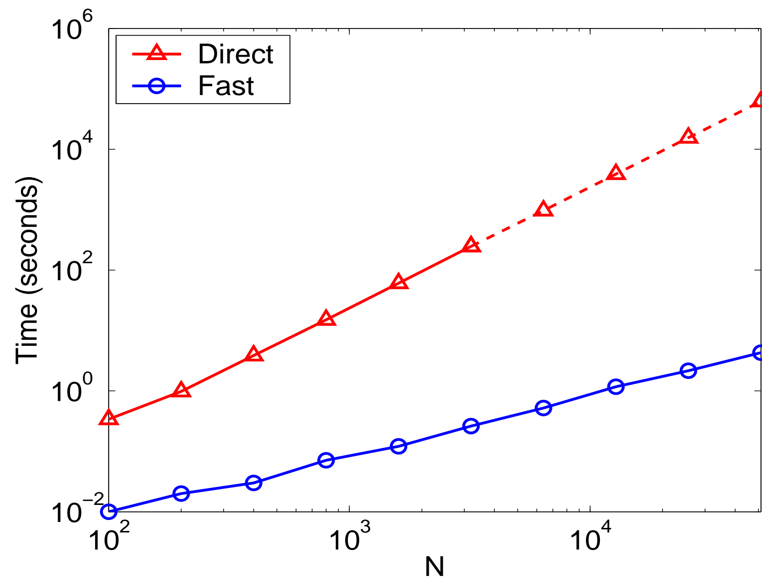
3. $h = \pi/3\left(r + \operatorname{erfc}^{-1}(\epsilon/2)\right)$.

4. $p = \left\lceil \frac{1}{2h}\operatorname{erfc}^{-1}\left(\frac{\sqrt{\pi}h\epsilon}{4}\right)\right\rceil$.

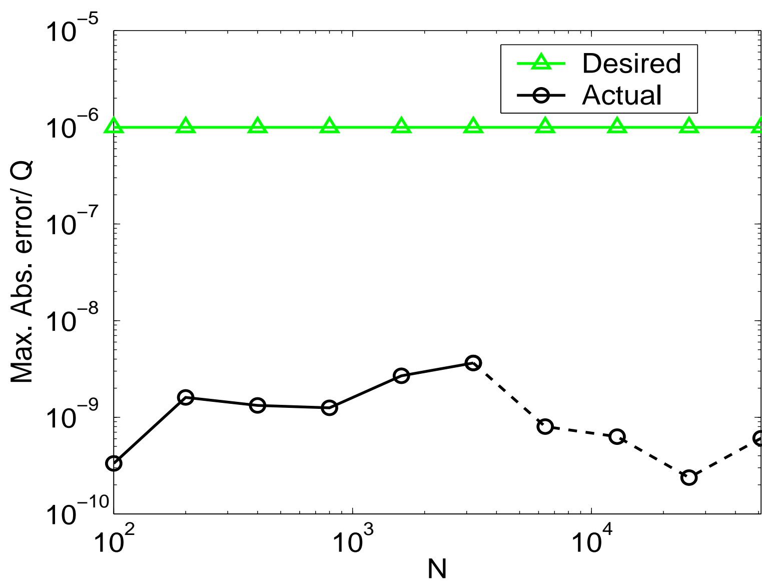
5. $(2l+1) = \lceil r/r_x \rceil$.

4.8 Numerical experiments

In this section we present some numerical studies of the speedup and error as a function of the number of data points and the desired error ϵ. The algorithm was programmed in C++ with MATLAB bindings and was run on a 1.6 GHz Pentium M processor with 512MB of RAM.

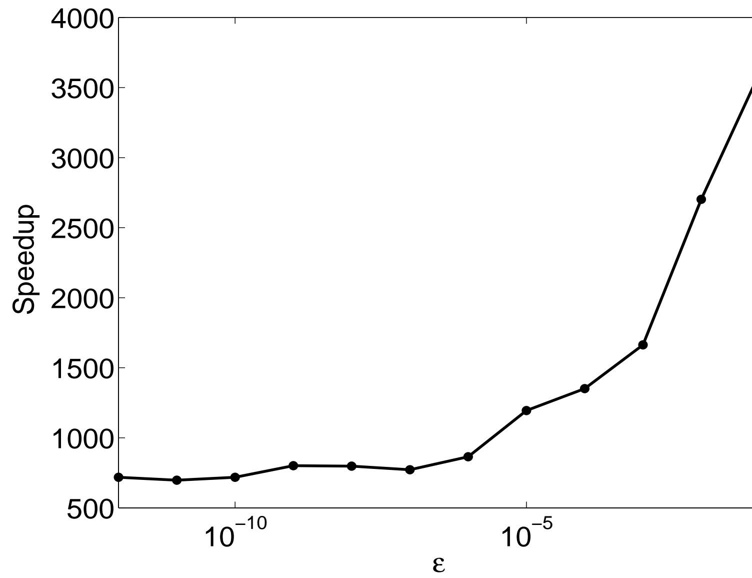


(a)

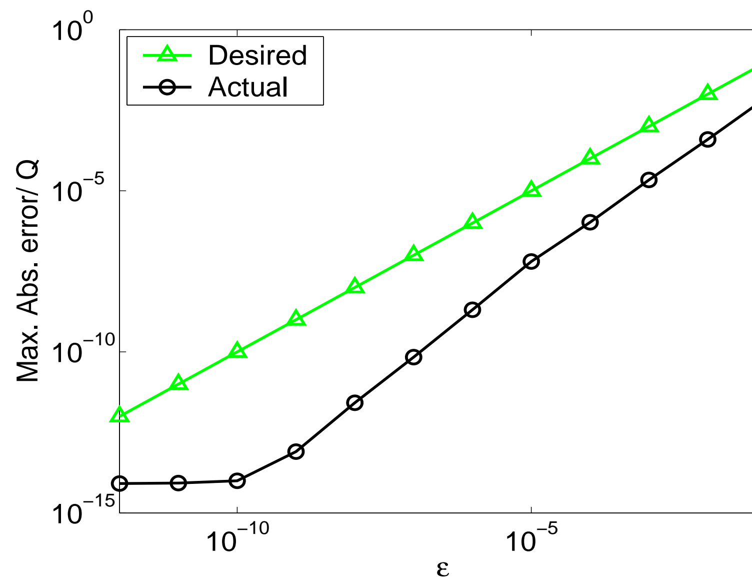


(b)

Figure 4.4: (a) The running time in seconds and (b) maximum absolute error relative to Q_{abs} for the direct and the fast methods as a function of $N = M$. For $N > 3,200$ the timing results for the direct evaluation were obtained by evaluating the sum at $M = 100$ points and then extrapolating (shown as dotted line).



(a)



(b)

Figure 4.5: (a) The speedup achieved and (b) maximum absolute error relative to Q for the direct and the fast methods as a function of ϵ for $N = M = 3,000$.

Figure 4.4 shows the running time and the maximum absolute error relative to Q_{abs} for both the direct and the fast methods as a function of $N = M$. The points were normally distributed with zero mean and unit variance. The weights q_i were set to 1. We see that the running time of the fast method grows linearly, while that of the direct evaluation grows quadratically. We also observe that the error is way below the desired error thus validating our bound. For example for $N = M = 51,200$ points while the direct evaluation takes around 17.26 hours the fast evaluation requires only 4.29 seconds with an error of around 10^{-10}. Figure 4.5 shows the tradeoff between precision and speedup. An increase in speedup is obtained at the cost of reduced accuracy.

Chapter 5

Kernel density estimation

Kernel density estimation [87] techniques are widely used in exploratory data analysis, various inference procedures in machine learning, data mining, pattern recognition, and computer vision.

A random variable X on $\mathbf{R}^d$ has a density p if, for all Borel sets A of $\mathbf{R}^d$, $\int_A p(x)dx = \Pr[x \in A]$. The task of density estimation is to estimate p from an i.i.d. sample $x_1, \ldots, x_N$ drawn from p. The estimate $\widehat{p} : \mathbf{R}^d \times \left(\mathbf{R}^d\right)^N \rightarrow \mathbf{R}$ is called the density estimate.

The parametric approach to density estimation assumes a functional form for the density, and then estimates the unknown parameters using techniques like the maximum likelihood estimation. However unless the form of the density is known a priori, assuming a functional form for a density very often leads to erroneous inference. On the other hand nonparametric methods do not make any assumption on the form of the underlying density. The price to be paid is a rate of convergence slower than $1/N$, which is typical of parametric methods. Some of the commonly used non-parametric estimators include histograms, kernel density estimators, and orthogonal series estimators [39].

5.1 Kernel density estimation

The most popular non-parametric method for density estimation is the kernel density estimator (KDE) (also known as the Parzen window estimator [54]). In its most general form , the d-dimensional KDE is

$$\widehat{p}(x) = \frac{1}{N}\sum_{i=1}^{N} K_{\mathbf{H}}\left(x - x_i\right), \text{ where } K_{\mathbf{H}}(x) = |\mathbf{H}|^{-1/2}K(\mathbf{H}^{-1/2}x). \tag{5.1}$$

The d-variate function K is called the *kernel function* and $\mathbf{H}$ is a symmetric positive definite $d \times d$ matrix called the *bandwidth matrix*. In order that $\widehat{p}(x)$ is a bona fide density, the kernel function is required to satisfy the following two conditions:

$$K(u) \geq 0, \text{ and } \int_{\mathbf{R^d}} K(u)du = 1. \tag{5.2}$$

The most commonly used kernel is the standard d-variate normal density–

$$K(u) = (2\pi)^{-d/2}e^{-\|u\|^2/2}. \tag{5.3}$$

In general a fully parameterized $d \times d$ positive definite bandwidth matrix $\mathbf{H}$ can be used to define the density estimate. However in high dimensions the number of independent parameters $(d(d+1)/2)$ are too large to make a good choice. Hence the most commonly used choice is $\mathbf{H} = diag(h_1^2, \ldots, h_d^2)$ or $\mathbf{H} = h^2\mathbf{I}$. For the case when $\mathbf{H} = h^2\mathbf{I}$ the density estimate can be written as,

$$\widehat{p}(x) = \frac{1}{N}\sum_{i=1}^{N} \frac{1}{(2\pi h^2)^{d/2}} e^{-\|x-x_i\|^2/2h^2}. \tag{5.4}$$

The computational cost of evaluating (6.5) at M points due to N data points is $\mathcal{O}(NM)$, making it prohibitively expensive for large datasets. The proposed

FIGTree algorithm (chapter 2) can be used to compute the sum approximately to ϵ precision in $\mathcal{O}(N+M)$ time. Given a specified precision ϵ it computes as approximation $\widehat{p}_{\mathrm{FIGTree}}(x)$ to $\widehat{p}(x)$ such that

$$|\widehat{p}(x) - \widehat{p}_{\mathrm{FIGTree}}(x)| \leq (2\pi h^2)^{-d/2}\epsilon. \tag{5.5}$$

5.2 Bandwidth selection

For a practical implementation of KDE the choice of the bandwidth h is very important. A small h leads to an estimator with small bias and large variance, while a large h leads to a small variance at the expense of an increase in the bias. The bandwidth h has to be chosen optimally. Various techniques have been proposed for optimal bandwidth selection [41]. They fall into broadly two categories– (1) plug-in bandwidths and (2) bandwidths chosen by cross-validation.

The plug-in bandwidths are known to show more stable performance [87] than the cross-validation methods. They are based on deriving an expression for the Asymptotic Mean Integrated Squared Error (AMISE) as a function of the bandwidth and then choosing the bandwidth which minimizes it. The simplest among these known as the *rules of thumb* (ROT) assumes that the data is generated by a multivariate normal distribution. For a normal distribution with covariance matrix $\Sigma = diag(\sigma_1^2, \ldots, \sigma_d^2)$ and the bandwidth matrix of the form $\mathbf{H} = diag(h_1^2, \ldots, h_d^2)$ the optimal bandwidths are given by [87]

$$h_j^{ROT} = \left(\frac{4}{d+2}\right)^{1/(d+4)} N^{-1/(d+4)}\widehat{\sigma}_j, \tag{5.6}$$

where $\widehat{\sigma}_j$ is an estimate of σ_j. This method is known to provide a quick first guess

and can be expected to give reasonable bandwidth when the data is close to a normal distribution. It is reported that this tends to slightly oversmooth the data. So in our experiments we only use this as a guess and show the speedup achieved over a range of bandwidths around h^{ROT}. As a practical issue we did not prefer cross-validation because we will have to do the KDE for a range of bandwidths, both small and large.

5.3 Experiments

For our experimental comparison we used the SARCOS dataset [1]. The dataset contains 44,484 samples in a 21 dimensional space. The data relates to an inverse dynamics problem for a seven degrees-of-freedom SARCOS anthropomorphic robot arm. We use the 21-dimensional input variables in order to perform the kernel density estimation. In order to ease comparisons all the dimensions were normalized to have the same variance so that we could use only one bandwidth parameter h.

Figure 5.1 compares the time taken by the direct summation, FIGTree, and the kd-tree based dual-tree method for different dimensions. In each of the plots the KDE was computed for the first d dimensions. The results are shown for $N = 7,000$ points so that the methods could be compared. The KDE was evaluated at $M = N$ points. The results are shown for a range of bandwidths around the optimal bandwidth obtained using the rule of thumb plug-in method. Accuracy of $\epsilon = 10^{-2}$ was used for all the methods. The FIGTree is faster than the dual-tree

[1] This dataset can be downloaded from the website `http://www.gaussianprocess.org/gpml/data/`.

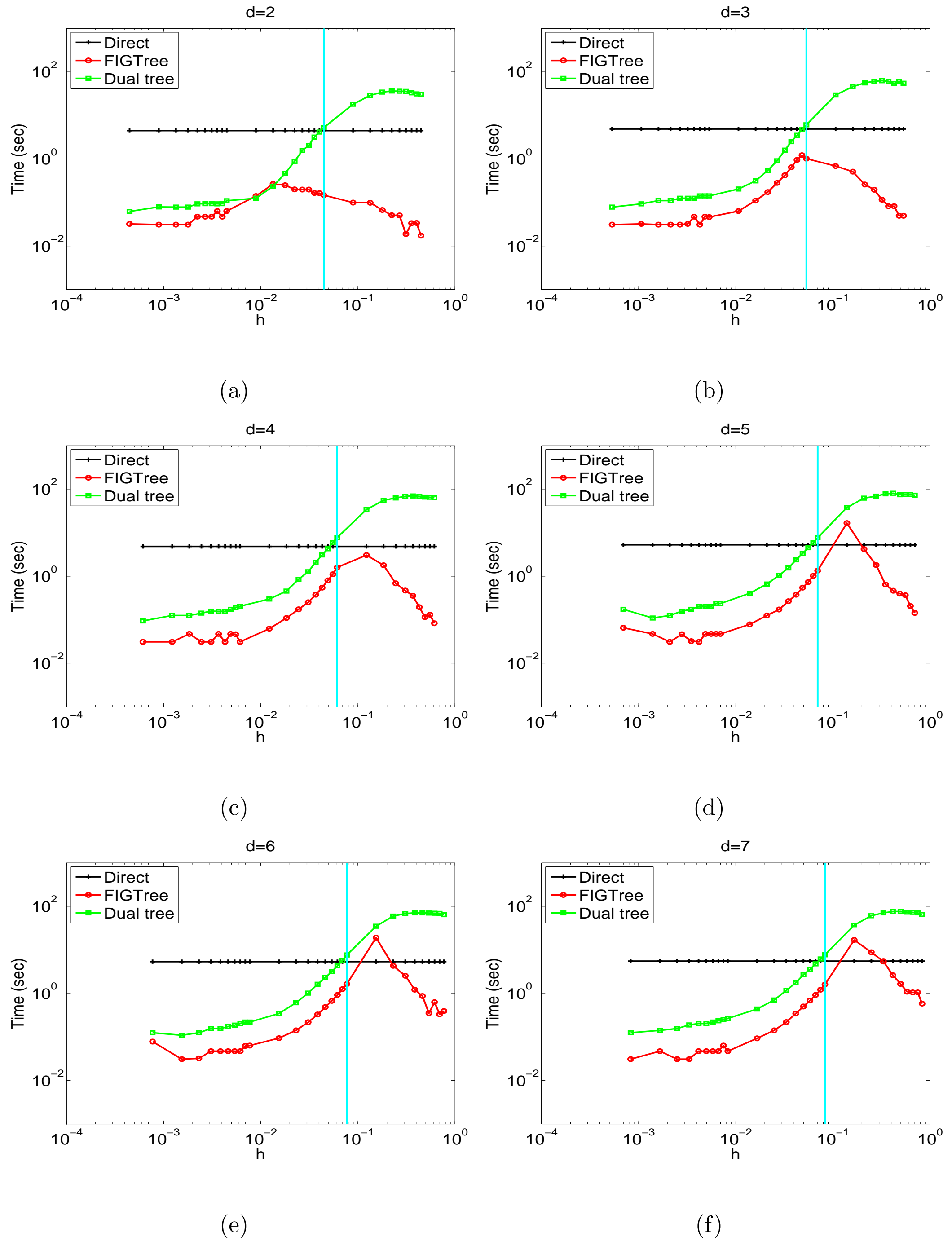


Figure 5.1: *KDE experiments for* $N = 7,000$ *source points.* The run time in seconds for the direct, FIGTree, and the dual-tree method for varying dimensionality, d. The results are shown for a range of bandwidths around the optimal bandwidth marked by the straight line in each of the plots. The error was set to $\epsilon = 10^{-2}$. The KDE was evaluated at $M = N$ points.

algorithm across all bandwidths. The dual-tree algorithm shows good performance only at very small bandwidths. At such bandwidths the FIGTree algorithm builds the kd-tree directly on the source points and is still faster than the dual tree method.

In the previous plot we used only $7,000$ points in order to compare our algorithm with the dual-tree method. Table 5.1 shows the time taken by the FIGTree algorithm on the entire dataset. The FIGTree algorithm gives good speedups for arbitrarily high dimensions for small and large bandwidths. However for moderate bandwidths and for $d > 7$ the FIGTRee was roughly twice as fast as the direct computation.

Table 5.1: *KDE experiments on the entire dataset.* Time taken by the direct summation and the FIGTree on the entire dataset containing $N = 44,484$ source points. The KDE was evaluated at $M = N$ points. The error was set to $\epsilon = 10^{-2}$.

d	Optimal h	Direct time (sec.)	FIGTree time (sec.)	Speedup
1	0.024730	168.500	0.110	1531.818
2	0.033357	180.156	0.844	213.455
3	0.041688	189.438	6.094	31.0860
4	0.049527	196.375	19.047	10.310
5	0.056808	208.453	97.156	2.146
6	0.063527	221.906	130.250	1.704
7	0.069711	226.375	121.829	1.858
8	0.075400	236.781	106.203	2.230
9	0.080637	247.235	88.250	2.801
10	0.085465	254.547	98.718	2.579

Chapter 6

Optimal bandwidth estimation

Most state-of-the-art automatic bandwidth selection procedures for kernel density estimates require estimation of quantities involving the density derivatives. The computational complexity of evaluating the density derivative at M evaluation points given N sample points from the density scales as $O(MN)$. In this paper we propose a computationally efficient $\epsilon - exact$ approximation algorithm for the univariate Gaussian kernel based density derivative estimation that reduces the computational complexity from $O(MN)$ to linear $O(M + N)$. The constant depends on the desired arbitrary accuracy, ϵ. For example for $N = M = 409,600$ points while the direct evaluation of the density derivative takes around 12.76 hours the fast evaluation requires only 65 seconds with an error of around 10^{-12}. We apply the density derivative evaluation procedure to estimate the optimal bandwidth for kernel density estimation, a process that is often intractable for large data sets. We demonstrate the speedup achieved on the bandwidth selection using the solve-the-equation plug-in method. For $50,000$ points sampled from the normal mixture densities of [49] we obtained speedups from 65 to 105. We also demonstrate that the proposed procedure can be extremely useful for speeding up exploratory projection pursuit techniques. [60, 61]

6.1 Introduction

Kernel density estimation [87] techniques are widely used in exploratory data analysis, various inference procedures in machine learning, data mining, pattern recognition, and computer vision. Efficient use of these methods require the optimal selection of the smoothing parameter called the *bandwidth* of the kernel. A plethora of techniques have been proposed for automatic data-driven bandwidth selection (see [41] for a review). The most successful state-of-the-art methods rely on the estimation of general integrated squared *density derivative functionals.* This is the most computationally intensive task, the computational cost being $O(N^2)$, where N is the number of sample points. The core task is to *efficiently compute an estimate of the density derivative.* The current most practically successful approach, *solve-the-equation plug-in* method of [72] involves the numerical solution of a non-linear equation. Iterative methods to solve this equation repeatedly use the density derivative functional estimator for different bandwidths which increases the computational burden. The estimation of the density derivative also comes up in various other applications like estimation of modes and inflexion points of densities [22] and estimation of the derivatives of the projection index in projection pursuit algorithms [38, 42]. A list of applications which require the estimation of density derivatives can be found in [76].

The computational complexity of evaluating the density derivative at M evaluation points given N sample points from the density is $O(MN)$. In this paper we propose a computationally efficient $\epsilon - exact$ approximation algorithm for the

univariate Gaussian kernel based density derivative estimation that reduces the computational complexity from $O(MN)$ to linear $O(N+M)$. The algorithm is $\epsilon - exact$ in the sense that the constant hidden in $O(N+M)$, depends on the desired *accuracy*, ϵ, which can be *arbitrary*. In fact for machine precision accuracy there is no difference in the answers provided by the direct and the fast methods.

The rest of the chapter is organized as follows. In § 6.2 we introduce the kernel density estimate and briefly review the asymptotic performance of the estimator. The concept of optimal bandwidth is introduced. The kernel density derivative estimate is introduced in § 6.3. In § 6.4 we discus the density derivative functionals which are used by most automatic bandwidth selection strategies. § 6.5 briefly describes the different strategies for automatic optimal bandwidth selection. The solve-the-equation plug-in method is described in detail. Our proposed fast method is described in detail in § 6.6. Algorithm details, error bounds, procedure to choose the parameters, and numerical experiments are presented. In § 6.7 we show the speedup achieved for bandwidth estimation both on simulated and real data. In § 6.8 we also show how the proposed procedure can be used for speeding up projection pursuit techniques. § 6.10 finally concludes with a brief discussion on further extensions.

6.2 Kernel Density Estimation

A univariate random variable X on $\mathbf{R}$ has a density p if, for all Borel sets A of $\mathbf{R}$,

$$\int_A p(x)dx = \Pr[x \in A]. \tag{6.1}$$

The task of density estimation is to estimate p from i.i.d. samples $x_1, \ldots, x_N$ drawn from p. The estimate $\widehat{p} : \mathbf{R} \times (\mathbf{R})^N \rightarrow \mathbf{R}$ is called the *density estimate.* The *parametric approach* to density estimation assumes a functional form for the density, and then estimates the unknown parameters using techniques like the maximum likelihood estimation. However unless the form of the density is known a priori, assuming a functional form for a density very often leads to erroneous inference. On the other hand *nonparametric methods* do not make any assumptions on the form of the underlying density. This is sometimes referred to as *'letting the data speak for themselves'* [87]. The price to be paid is a rate of convergence slower than $1/N$, which is typical of parametric methods. Some of the commonly used non-parametric estimators include histograms, kernel density estimators, and orthogonal series estimators [39].

The most popular non-parametric method for density estimation is the *kernel density estimator* (KDE) (also known as the *Parzen window estimator* [54]) given by

$$\widehat{p}(x) = \frac{1}{Nh} \sum_{i=1}^{N} K\left(\frac{x - x_i}{h}\right), \tag{6.2}$$

where K is called the *kernel function* and $h \in \mathbf{R}^+$ is called the *bandwidth* of the kernel. The bandwidth h is a scaling factor which goes to zero as $N \rightarrow 0$. In order

that $\widehat{p}(x)$ is a bona fide density, K is required to satisfy the following two conditions:

$$K(u) \geq 0, \quad \int_{\mathbf{R}} K(u)du = 1. \tag{6.3}$$

The kernel function is essentially spreading a probability mass of $1/N$ associated with each point about its neighborhood. The most widely used kernel is the Gaussian of zero mean and unit variance.

$$K(u) = \frac{1}{\sqrt{2\pi}} e^{-u^2/2}. \tag{6.4}$$

For the Gaussian kernel the kernel density estimate can be written as

$$\widehat{p}(x) = \frac{1}{N\sqrt{2\pi h^2}} \sum_{i=1}^{N} e^{-(x-x_i)^2/2h^2}. \tag{6.5}$$

The KDE is not very sensitive to the shape of the kernel. While the Epanechnikov kernel is the optimal kernel, in the sense that it minimizes the MISE, other kernels are not that suboptimal [87]. The Epanechnikov kernel is not used here because it gives an estimate having a discontinuous first derivative, because of its finite support.

The computational cost of evaluating Eq. 6.5 at M points is $O(MN)$, making it prohibitively expensive for large data sets. Different methods [74, 29, 96, 26] have been proposed to accelerate this sum. The main contribution of this paper is to accelerate the kernel *density derivative* estimate, and solve the optimal bandwidth problem.

6.2.1 AMISE optimal bandwidth for kernel density estimate

In order to understand the performance of the KDE we need a measure of distance between two densities. The commonly used criteria, which can be easily manipulated is the L_2 norm, also called as the *integrated square error* (ISE). The ISE between the estimate $\widehat{p}(x)$ and the actual density $p(x)$ is given by

$$\text{ISE}(\widehat{p}, p) = L_2(\widehat{p}, p) = \int_{\mathbf{R}} [\widehat{p}(x) - p(x)]^2 dx. \tag{6.6}$$

The ISE depends on a particular realization of N points. The ISE can be averaged over these realizations to get the *mean integrated squared error* (MISE) defined as

$$\begin{aligned} \text{MISE}(\widehat{p}, p) &= E[\text{ISE}(\widehat{p}, p)] = E\left[\int_{\mathbf{R}} [\widehat{p}(x) - p(x)]^2 dx\right] \\ &= \int_{\mathbf{R}} E[\{\widehat{p}(x) - p(x)\}^2] dx = \text{IMSE}(\widehat{p}, p), \end{aligned} \tag{6.7}$$

where IMSE is *integrated mean squared error*. The MISE or IMSE doesn't depend on the actual data-set as we take expectation. So this is a measure of the 'average' performance of the kernel density estimator, averaged over the support of the density and different realization of the points. The MISE for the KDE can be shown to be (see [87] for a derivation)

$$\text{MISE}(\widehat{p}, p) = \frac{1}{N} \int_{\mathbf{R}} \left[(K_h^2 * p)(x) - (K_h * p)^2(x)\right] dx + \int_{\mathbf{R}} \left[(K_h * p)(x) - p(x)\right]^2 dx, \tag{6.8}$$

where $*$ is the convolution operator and $K_h(x) = (1/h)K(x/h)$. The dependence of the MISE on the bandwidth h is not very explicit in the above expression. This makes it difficult to interpret the influence of the bandwidth on the performance

of the estimator. An asymptotic large sample approximation for this expression is usually derived via the Taylor's series called as the AMISE, the A is for asymptotic. Based on certain assumptions [1], the AMISE between the actual density and the estimate can be shown to be (See [87] for a complete derivation.)

$$\text{AMISE}(\widehat{p}, p) = \frac{1}{Nh} R(K) + \frac{1}{4} h^4 \mu_2(K)^2 R(p''), \tag{6.9}$$

where

$$R(g) = \int_{\mathbf{R}} g(x)^2 dx \ , \ \mu_2(g) = \int_{\mathbf{R}} x^2 g(x) dx, \tag{6.10}$$

and p'' is the second derivative of the density p . The first term in the expression 6.9 is the integrated variance and the second term is the integrated squared bias. The squared bias is proportional to h^4 whereas the variance is proportional to $1/h$, which leads to the well known *bias-variance tradeoff*.

For a practical implementation of KDE the choice of the bandwidth h is very important. A small h leads to an estimator with small bias and large variance, while a large h leads to a small variance at the expense of an increase in the bias. The bandwidth h has to be chosen optimally. Various techniques have been proposed for optimal bandwidth selection. Most of them are based on the AMISE.

Based on the AMISE expression the optimal bandwidth h_{AMISE} can be obtained by differentiating Eq. 6.9 with respect to the bandwidth h and setting it to

[1]The second derivative $p''(x)$ is continuous, square integrable, and ultimately monotone. $\lim_{N\to\infty} h = 0$ and $\lim_{N\to\infty} Nh = \infty$, i.e., as the number of samples N is increased h approaches zero at a rate slower than $1/N$.

zero, to obtain

$$h_{AMISE} = \left[\frac{R(K)}{\mu_2(K)^2 R(p'') N} \right]^{1/5}. \tag{6.11}$$

However this expression cannot be used directly since $R(p^{''})$ depends on the second derivative of the density p, which we are trying to estimate in the first place. We need to use an estimate of $R(p^{''})$.

6.3 Kernel Density Derivative estimation

In order to estimate $R(p^{''})$ we will need an estimate of the density derivative. A simple estimator for the density derivative can be obtained by taking the derivative of the kernel density estimate $\widehat{p}(x)$ defined earlier [6, 70]. If the kernel K is differentiable r times then the rth density derivative estimate $\widehat{p}^{(r)}(x)$ can be written as

$$\widehat{p}^{(r)}(x) = \frac{1}{Nh^{r+1}} \sum_{i=1}^{N} K^{(r)} \left(\frac{x - x_i}{h} \right), \tag{6.12}$$

where $K^{(r)}$ is the rth derivative of the kernel K. The rth derivative of the Gaussian kernel $k(u)$ is given by

$$K^{(r)}(u) = (-1)^r H_r(u) K(u), \tag{6.13}$$

where $H_r(u)$ is the rth Hermite polynomial. The Hermite polynomials are a set of orthogonal polynomials [1] . The first few Hermite polynomials are

$$H_0(u) \quad = \quad 1,\ H_1(u) = u, \text{ and } H_2(u) = u^2 - 1.$$

Hence the density derivative estimate with the Gaussian kernel can be written as

$$\widehat{p}^{(r)}(x) = \frac{(-1)^r}{\sqrt{2\pi} N h^{r+1}} \sum_{i=1}^{N} H_r \left(\frac{x - x_i}{h} \right) e^{-(x-x_i)^2/2h^2}. \tag{6.14}$$

The computational complexity of evaluating the rth derivative of the density estimate from N points at M target locations is thus $O(rNM)$.

6.3.1 AMISE optimal bandwidth for kernel density derivative estimate

The optimal bandwidth for estimating the kernel density derivative is not the same as that used for KDE. Similar to the analysis done for KDE the AMISE for the kernel density derivative estimate, under certain assumptions [2] can be shown to be

$$\text{AMISE}(\widehat{p}^{(r)}, p^{(r)}) = \frac{R(K^{(r)})}{Nh^{2r+1}} + \frac{h^4}{4}\mu_2(K)^2 R(p^{(r+2)}). \tag{6.15}$$

Differentiating Eq. 6.15 w.r.t. bandwidth h and setting it to zero we obtain the optimal bandwidth h^r_{AMISE} to estimate the rth density derivative.

$$h^r_{AMISE} = \left[\frac{R(K^{(r)})(2r+1)}{\mu_2(K)^2 R(p^{(r+2)})N}\right]^{1/2r+5}. \tag{6.16}$$

It can be observed that the AMISE optimal bandwidth for estimating the rth derivative depends upon the the $(r+2)^{th}$ derivative of the true density.

6.4 Estimation of Density Functionals

Rather than the actual density derivative methods for automatic bandwidth selection require the estimation of what are known as *density functionals*. The

[2] The $(r+2)^{th}$ derivative $p^{(r+2)}(x)$ is continuous, square integrable and ultimately monotone. $\lim_{N\to\infty} h = 0$ and $\lim_{N\to\infty} Nh^{2r+1} = \infty$, i.e., as the number of samples N is increased h approaches zero at a rate slower than $1/N^{2r+1}$.

general integrated squared density derivative functional is defined as

$$R(p^{(s)}) = \int_{\mathbf{R}} \left[p^{(s)}(x)\right]^2 dx. \tag{6.17}$$

Using integration by parts, this can be written in the following form [87],

$$R(p^{(s)}) = (-1)^s \int_{\mathbf{R}} p^{(2s)}(x)p(x)dx. \tag{6.18}$$

More specifically for even s we are interested in estimating density functionals of the form,

$$\Phi_r = \int_{\mathbf{R}} p^{(r)}(x)p(x)dx = E\left[p^{(r)}(X)\right]. \tag{6.19}$$

An estimator for Φ_r is,

$$\widehat{\Phi}_r = \frac{1}{N}\sum_{i=1}^{N} \widehat{p}^{(r)}(x_i). \tag{6.20}$$

where $\widehat{p}^{(r)}(x_i)$ is the estimate of the rth derivative of the density $p(x)$ at $x = x_i$. Using a kernel density derivative estimate for $\widehat{p}^{(r)}(x_i)$ (Eq. 6.12) we have

$$\widehat{\Phi}_r = \frac{1}{N^2 h^{r+1}} \sum_{i=1}^{N}\sum_{j=1}^{N} K^{(r)}\left(\frac{x_i - x_j}{h}\right). \tag{6.21}$$

It should be noted that computation of $\widehat{\Phi}_r$ is $O(rN^2)$ and hence can be very expensive if a direct algorithm is used.

6.4.1 AMSE optimal bandwidth for density functional estimation

The asymptotic MSE for the density functional estimator under certain assumptions [3] is as follows.

$$\begin{aligned}\text{AMSE}(\widehat{\Phi}_r, \Phi_r) &= \left[\frac{1}{Nh^{r+1}}K^{(r)}(0) + \frac{1}{2}h^2\mu_2(K)\Phi_{r+2}\right]^2 + \frac{2}{N^2h^{2r+1}}\Phi_0 R(K^{(r)}) \\ &\quad + \frac{4}{N}\left[\int p^{(r)}(y)^2 p(y)dy - \Phi_r^2\right] \end{aligned} \tag{6.22}$$

(see [87] for a complete derivation.). The optimal bandwidth for estimating the density functional is chosen the make the bias term zero. The optimal bandwidth is given by [87]

$$g_{\text{MSE}} = \left[\frac{-2K^{(r)}(0)}{\mu_2(K)\Phi_{r+2}N}\right]^{1/r+3}. \tag{6.23}$$

6.5 AMISE optimal Bandwidth Selection

Based on the AMISE expression the optimal bandwidth h_{AMISE} has the following form,

$$h_{\text{AMISE}} = \left[\frac{R(K)}{\mu_2(K)^2 R(p'')N}\right]^{1/5}. \tag{6.24}$$

However this expression cannot be used directly since $R(p'')$ depends on the second derivative of the density p, which we are trying to estimate in the first place.

[3] The density p had $k > 2$ continuous derivatives which are ultimately monotone. The $(r+2)^{th}$ derivative $p^{(r+2)}(x)$ is continuous, square integrable and ultimately monotone. $\lim_{N\to\infty} h = 0$ and $\lim_{N\to\infty} Nh^{2r+1} = \infty$, i.e., as the number of samples N is increased h approaches zero at a rate slower than $1/N^{2r+1}$.

6.5.1 Review of different methods

Different strategies have been proposed to solve this problem. A brief survey can be found in [41] and [87]. The best known of these include rules of thumb, oversmoothing, least squares cross-validation, biased cross-validation, direct plug-in methods, solve-the-equation plug-in method, and the smoothed bootstrap.

The *rules of thumb* use an estimate of $R(p^{''})$ assuming that the data is generated by some parametric form of the density (typically a normal distribution). The *oversmoothing* methods rely on the fact that there is a simple upper bound for the AMISE-optimal bandwidth for estimation of densities with a fixed value of a particular scale measure. The *least squares cross-validation* directly minimize the MISE based on a leave-one-out kernel density estimator. The problem is that the function to be minimized has fairly large number of local minima and also the practical performance of this method is somewhat disappointing. The *biased cross-validation* uses the AMISE instead of using the exact MISE formula. This is more stable than the least squares cross-validation but has a large bias.

The *plug-in methods* use an estimate of the density functional $R(p^{''})$ in Eq. 6.24. However this is not completely automatic since estimation of $R(p^{''})$ requires the specification of another *pilot bandwidth* g. This bandwidth for estimation of the density functional is quite different from the the bandwidth h used for the kernel density estimate. As discussed in Section 6.4 we can find an expression for the optimal bandwidth for the estimation of $R(p^{''})$. However this bandwidth will depend on an unknown density functional $R(p^{'''})$. This problem will continue since the optimal

bandwidth for estimating $R(p^{(s)})$ will depend on $R(p^{(s+1)})$. The usual strategy used by the *direct plug-in* methods is to estimate $R(p^{(l)})$ for some l, with bandwidth chosen with reference to a parametric family, usually a normal density. This method is usually referred to as the *l-stage direct plug-in* method. As the the number of stages l increases the bias of the bandwidth decreases, since the dependence on the assumption of some parametric family decreases. However this comes at the price of the estimate being more variable. There is no good method for the choice of l, the most common choice being $l = 2$.

The most successful among all the current methods, both empirically and theoretically, is the *solve-the-equation plug-in* method [41]. This method differs from the direct plug-in approach in that the pilot bandwidth used to estimate the density functional $R(p'')$ is written as a function of the kernel bandwidth h used to estimate the density. We use the following version as described in [72].

6.5.2 Solve-the-equation plug-in method

The AMISE optimal bandwidth is the solution to the equation

$$h = \left[\frac{R(K)}{\mu_2(K)^2 \widehat{\Phi}_4[\gamma(h)] N} \right]^{1/5}, \tag{6.25}$$

where $\widehat{\Phi}_4[\gamma(h)]$ is an estimate of $\Phi_4 = R(p'')$ using the pilot bandwidth $\gamma(h)$, which depends on the kernel bandwidth h. The bandwidth is chosen such that it minimizes the asymptotic MSE for the estimation of Φ_4 and is given by (substituting $r = 4$ in Eq. 6.23)

$$g_{\text{MSE}} = \left[\frac{-2K^{(4)}(0)}{\mu_2(K) \Phi_6 N} \right]^{1/7}. \tag{6.26}$$

Substituting for N from Eq. 6.24 g_{MSE} can be written as a function of h as follows

$$g_{\text{MSE}} = \left[\frac{-2K^{(4)}(0)\mu_2(K)\Phi_4}{R(K)\Phi_6} \right]^{1/7} h_{\text{AMISE}}^{5/7}. \tag{6.27}$$

This suggest that we set

$$\gamma(h) = \left[\frac{-2K^{(4)}(0)\mu_2(K)\widehat{\Phi}_4(g_1)}{R(K)\widehat{\Phi}_6(g_2)} \right]^{1/7} h^{5/7}, \tag{6.28}$$

where $\widehat{\Phi}_4(g_1)$ and $\widehat{\Phi}_6(g_2)$ are estimates of Φ_4 and Φ_6 using bandwidths g_1 and g_2 respectively.

$$\widehat{\Phi}_4(g_1) = \frac{1}{N(N-1)g_1^5} \sum_{i=1}^{N} \sum_{j=1}^{N} K^{(4)} \left(\frac{x_i - x_j}{g_1} \right). \tag{6.29}$$

$$\widehat{\Phi}_6(g_2) = \frac{1}{N(N-1)g_2^7} \sum_{i=1}^{N} \sum_{j=1}^{N} K^{(6)} \left(\frac{x_i - x_j}{g_2} \right). \tag{6.30}$$

The bandwidths g_1 and g_2 are chosen to minimize the asymptotic MSE.

$$g_1 = \left[\frac{-2K^{(4)}(0)}{\mu_2(K)\widehat{\Phi}_6 N} \right]^{1/7} \quad g_2 = \left[\frac{-2K^{(6)}(0)}{\mu_2(K)\widehat{\Phi}_8 N} \right]^{1/9}, \tag{6.31}$$

where $\widehat{\Phi}_6$ and $\widehat{\Phi}_8$ are estimators for Φ_6 and Φ_8 respectively. We can use a similar strategy for estimation of Φ_6 and Φ_8. However this problem will continue since the optimal bandwidth for estimating Φ_r will depend on Φ_{r+2}. The usual strategy is to estimate a Φ_r at some stage, using a quick and simple estimate of bandwidth chosen with reference to a parametric family, usually a normal density. It has been observed that as the the number of stages increases the variance of the bandwidth increases. The most common choice is to use only two stages.

If p is a normal density with variance σ^2 then for even r we can compute Φ_r exactly [87].

$$\Phi_r = \frac{(-1)^{r/2} r!}{(2\sigma)^{r+1}(r/2)!\pi^{1/2}}. \tag{6.32}$$

An estimator of Φ_r will use an estimate $\widehat{\sigma}^2$ of the variance. Based on this we can write an estimator for Φ_6 and Φ_8 as follows.

$$\widehat{\Phi}_6 = \frac{-15}{16\sqrt{\pi}}\widehat{\sigma}^{-7} \text{and } \widehat{\Phi}_8 = \frac{105}{32\sqrt{\pi}}\widehat{\sigma}^{-9}. \qquad (6.33)$$

In this paper we use the Gaussian kernel for all estimates. The *two stage* solve-the-equation method using the *Gaussian kernel* can be summarized as follows.

1. Compute an estimate $\widehat{\sigma}$ of the standard deviation σ.

2. Estimate the density functionals Φ_6 and Φ_8 using the normal scale rule.

$$\widehat{\Phi}_6 = \frac{-15}{16\sqrt{\pi}}\widehat{\sigma}^{-7} \text{ and } \widehat{\Phi}_8 = \frac{105}{32\sqrt{\pi}}\widehat{\sigma}^{-9}.$$

3. Estimate the density functionals Φ_4 and Φ_6 using the kernel density derivative estimators with the optimal bandwidth based on the asymptotic MSE.

$$g_1 = \left[\frac{-6}{\sqrt{2\pi}\widehat{\Phi}_6 N}\right]^{1/7} \quad g_2 = \left[\frac{30}{\sqrt{2\pi}\widehat{\Phi}_8 N}\right]^{1/9}$$

$$\widehat{\Phi}_4(g_1) = \frac{1}{N(N-1)\sqrt{2\pi}g_1^5}\sum_{i=1}^{N}\sum_{j=1}^{N} H_4\left(\frac{x_i - x_j}{g_1}\right) e^{-(x_i-x_j)^2/2g_1^2}.$$

$$\widehat{\Phi}_6(g_2) = \frac{1}{N(N-1)\sqrt{2\pi}g_2^7}\sum_{i=1}^{N}\sum_{j=1}^{N} H_6\left(\frac{x_i - x_j}{g_2}\right) e^{-(x_i-x_j)^2/2g_2^2}.$$

4. The bandwidth is the solution to the equation

$$h - \left[\frac{1}{2\sqrt{\pi}\widehat{\Phi}_4[\gamma(h)]N}\right]^{1/5} = 0,$$

where

$$\widehat{\Phi}_4[\gamma(h)] = \frac{1}{N(N-1)\sqrt{2\pi}\gamma(h)^5}\sum_{i=1}^{N}\sum_{j=1}^{N} H_4\left(\frac{x_i - x_j}{\gamma(h)}\right) e^{-(x_i-x_j)^2/2\gamma(h)^2},$$

and

$$\gamma(h) = \left[\frac{-6\sqrt{2}\widehat{\Phi}_4(g_1)}{\widehat{\Phi}_6(g_2)} \right]^{1/7} h^{5/7}.$$

The last equation can be solved using any numerical routine like the Newton-Raphson method. The main computational bottleneck is the estimation of Φ which is of $O(N^2)$. Also note that solution to this equation will involve repeated use of the density derivative functional estimator for different bandwidths which adds further to the computational burden.

6.6 Fast $\epsilon - exact$ density derivative estimation

The computational cost of estimating the optimal bandwidth is $O(N^2)$. The core computational task contributing to this quadratic complexity is due to the computation of the kernel density derivative estimate at each of the N points.

The rth kernel density derivative estimate using the Gaussian kernel of bandwidth h is given by

$$\widehat{p}^{(r)}(x) = \frac{(-1)^r}{\sqrt{2\pi}Nh^{r+1}} \sum_{i=1}^{N} H_r\left(\frac{x - x_i}{h}\right) e^{-(x-x_i)^2/2h^2}. \tag{6.34}$$

Let us say we have to estimate the density derivative at M *target points*, $\{y_j \in \mathbf{R}\}_{j=1}^{M}$. More generally we need to evaluate the following sum,

$$G_r(y_j) = \sum_{i=1}^{N} q_i H_r\left(\frac{y_j - x_i}{h_1}\right) e^{-(y_j-x_i)^2/h_2^2} \quad j = 1, \ldots, M, \tag{6.35}$$

where $\{q_i \in \mathbf{R}\}_{i=1}^{N}$ will be referred to as the *source weights*, $h_2 \in \mathbf{R}^+$ is the bandwidth of the Gaussian and $h_1 \in \mathbf{R}^+$ will be referred to as the bandwidth of the Hermite.

The computational complexity of evaluating Eq. 6.35 is $O(rNM)$. In Chapter 3 we presented an $\epsilon - exact$ approximation algorithm that reduces the computational complexity to $O(prN + npr^2M)$, where the constants p and n depends on the precision ϵ and the bandwidth h. For any given $\epsilon > 0$ the algorithm computes an approximation $\widehat{G}_r(y_j)$ such that

$$\left| \frac{\hat{G}_r(y_j) - G_r(y_j)}{Q} \right| \leq \epsilon, \tag{6.36}$$

where $Q = \sum_{i=1}^{N} |q_i|$. We call $\widehat{G}_r(y_j)$ an $\epsilon - exact$ approximation to $G_r(y_j)$. The fast algorithm is based on separating the x_i and y_j in the Gaussian via the factorization of the Gaussian by Taylor series and retaining only the first few terms so that the error due to truncation is less than the desired error. The Hermite function is factorized via the binomial theorem.

6.7 Speedup achieved for bandwidth estimation

The solve-the-equation plug-in method of [41] was implemented in MATLAB with the core computational task of computing the density derivative written in C++.

6.7.1 Synthetic data

We demonstrate the speedup achieved on the mixture of normal densities used by [49]. The family of normal mixture densities is extremely rich and, in fact any density can be approximated arbitrarily well by a member of this family. Fig. 6.1 shows the fifteen densities which were used by the authors in [49] as a typical

representative of the densities likely to be encountered in real data situations. We sampled $N = 50,000$ points from each density. The AMISE optimal bandwidth was estimated both using the direct methods and the proposed fast method. For the fast method we used $\epsilon = 10^{-3}$. Table 6.1 shows the speedup achieved and the absolute relative error. The absolute relative error is defined as $|h_{direct} - h_{fast}/h_{direct}|$. We obtained speedups in the range 65 to 105 with the absolute relative error of the order 10^{-5} to 10^{-7}. Better speedups can be achieved by further increasing ϵ. Fig. 6.1 shows the actual density and the estimated density using the optimal bandwidth estimated using the fast method.

6.7.2 Real data

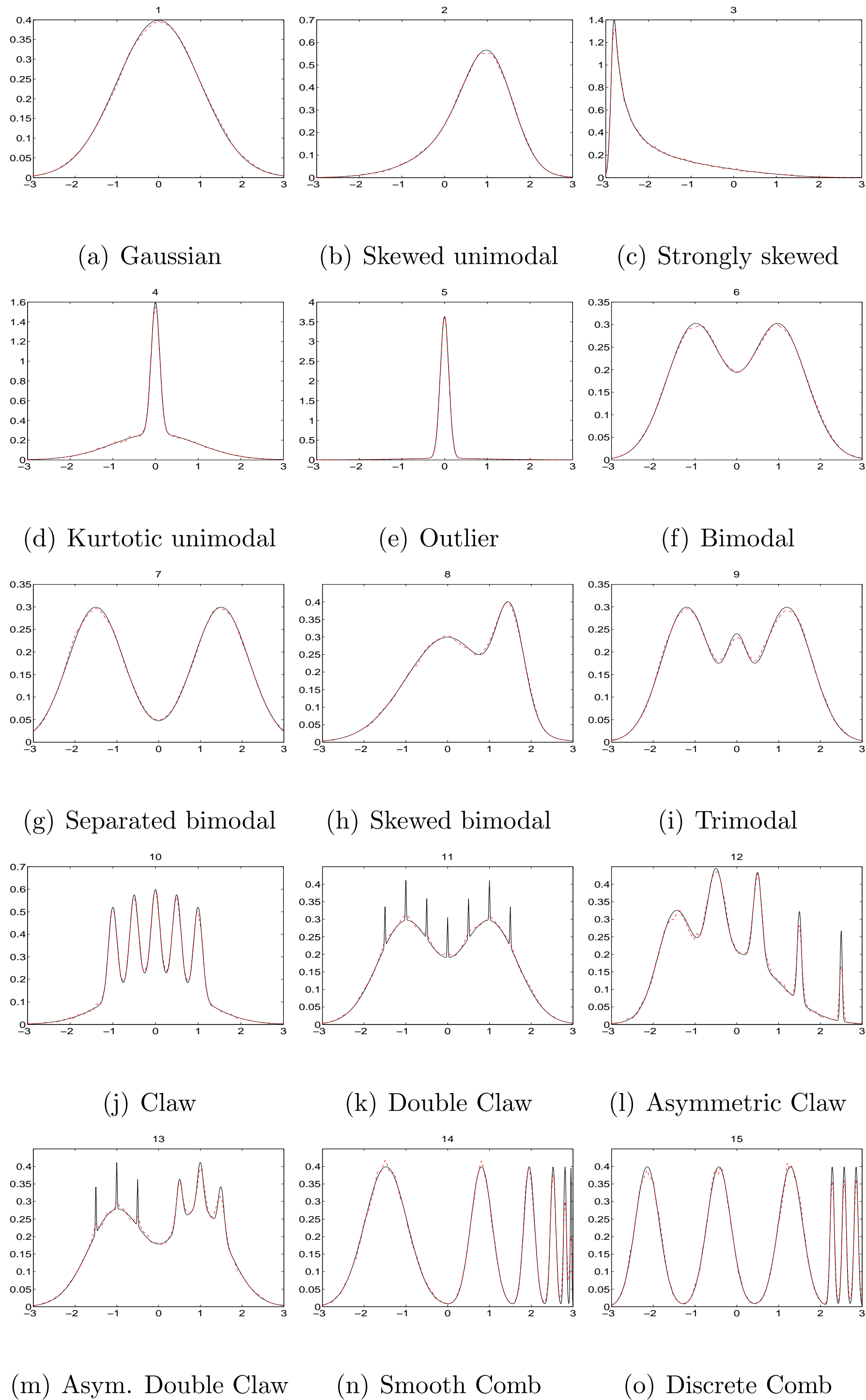
We used the Adult database from the UCI machine learning repository [52]. The database extracted from the census bureau database contains 32,561 training instances with 14 attributes per instance. Of the 14 attributes 6 are continuous and 8 nominal. Table 6.2 shows the speedup achieved and the absolute relative error for five of the continuous attributes. Fig. 6.2 shows the actual density and the estimated density for two attributes.

6.8 Projection Pursuit

Projection Pursuit (PP) is an exploratory technique for visualizing and analyzing large multivariate data-sets [21, 38, 42]. The idea of projection pursuit is to search for projections from high- to low-dimensional space that are most *interesting*.

Table 6.1: The bandwidth estimated using the solve-the-equation plug-in method for the fifteen normal mixture densities of Marron and Wand. h_{direct} and h_{fast} are the bandwidths estimated using the direct and the fast methods respectively. The running time in seconds for the direct and the fast methods are shown. The absolute relative error is defined as $|h_{direct} - h_{fast}/h_{direct}|$. In the study $N = 50,000$ points were sampled from the corresponding densities. For the fast method we used $\epsilon = 10^{-3}$.

Density	h_{direct}	h_{fast}	T_{direct} (sec)	T_{fast} (sec)	Speedup	Abs. Relative Error
1	0.122213	0.122215	4182.29	64.28	65.06	1.37e-005
2	0.082591	0.082592	5061.42	77.30	65.48	1.38e-005
3	0.020543	0.020543	8523.26	101.62	83.87	1.53e-006
4	0.020621	0.020621	7825.72	105.88	73.91	1.81e-006
5	0.012881	0.012881	6543.52	91.11	71.82	5.34e-006
6	0.098301	0.098303	5023.06	76.18	65.93	1.62e-005
7	0.092240	0.092240	5918.19	88.61	66.79	6.34e-006
8	0.074698	0.074699	5912.97	90.74	65.16	1.40e-005
9	0.081301	0.081302	6440.66	89.91	71.63	1.17e-005
10	0.024326	0.024326	7186.07	106.17	67.69	1.84e-006
11	0.086831	0.086832	5912.23	90.45	65.36	1.71e-005
12	0.032492	0.032493	8310.90	119.02	69.83	3.83e-006
13	0.045797	0.045797	6824.59	104.79	65.13	4.41e-006
14	0.027573	0.027573	10485.48	111.54	94.01	1.18e-006
15	0.023096	0.023096	11797.34	112.57	104.80	7.05e-007



(a) Gaussian (b) Skewed unimodal (c) Strongly skewed

(d) Kurtotic unimodal (e) Outlier (f) Bimodal

(g) Separated bimodal (h) Skewed bimodal (i) Trimodal

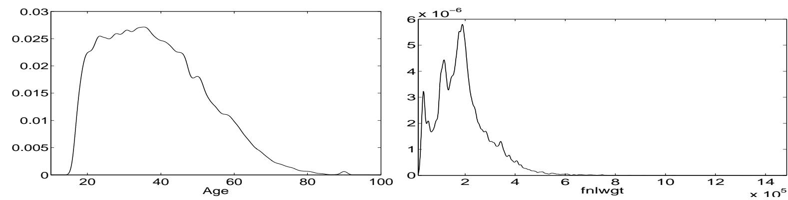
(j) Claw (k) Double Claw (l) Asymmetric Claw

(m) Asym. Double Claw (n) Smooth Comb (o) Discrete Comb

Figure 6.1: The fifteen normal mixture densities of Marron and Wand. The solid line corresponds to the actual density while the dotted line is the estimated density using the optimal bandwidth estimated using the fast method.

Table 6.2: Optimal bandwidths for the five continuous attributes for the Adult database from the UCI machine learning repository. The database contains $32,561$ training instances. The bandwidth was estimated using the solve-the-equation plug-in method. h_{direct} and h_{fast} are the bandwidths estimated using the direct and the fast methods respectively. The running time in seconds for the direct and the fast methods are shown. The absolute relative error is defined as $|h_{direct} - h_{fast}/h_{direct}|$. For the fast method we used $\epsilon = 10^{-3}$.

Attribute	h_{direct}	h_{fast}	T_{direct} (sec)	T_{fast} (sec)	Speedup	Error
Age	0.860846	0.860856	4679.03	66.42	70.45	1.17e-005
fnlwgt	4099.564359	4099.581141	4637.09	68.83	67.37	4.09e-006
capital-gain	2.376596	2.376596	7662.48	74.46	102.91	4.49e-010
capital-loss	0.122656	0.122656	7466.54	72.88	102.46	2.99e-011
hours-per-week	0.009647	0.009647	9803.80	130.37	75.20	2.27e-008



(a) Age (b) fnlwgt

Figure 6.2: The estimated density using the optimal bandwidth estimated using the fast method, for two of the continuous attributes in the Adult database from the UCI machine learning repository.

These projections can then be used for other nonparametric fitting and other data-analytic purposes The conventional dimension reduction techniques like principal component analysis looks for a projection that maximizes the variance. The idea of PP is to look for projections that maximize other measures of interestingness, like non-normality, entropy etc. The PP algorithm for finding the most interesting one-dimensional subspace is as follows.

1. Given N data points in a d dimensional space (centered and scaled), $\{x_i \in \mathbf{R}^d\}_{i=1}^N$, project each data point onto the direction vector $a \in \mathbf{R}^d$, i.e., $z_i = a^T x_i$.

2. Compute the univariate nonparametric kernel density estimate, $\widehat{p}$, of the projected points z_i.

3. Compute the projection index $I(a)$ based on the density estimate.

4. Locally optimize over the the choice of a, to get the *most interesting* projection of the data.

5. Repeat from a new initial projection to get a different view.

The projection index is designed to reveal specific structure in the data, like clusters, outliers, or smooth manifolds. Some of the commonly used projection indices are the Friedman-Tukey index [21], the entropy index [42], and the moment index. The entropy index based on Rényi's order-1 entropy is given by

$$I(a) = \int p(z) \log p(z) dz. \tag{6.37}$$

The density of zero mean and unit variance which uniquely minimizes this is the standard normal density. Thus the projection index finds the direction which is most non-normal. In practice we need to use an estimate $\widehat{p}$ of the the true density p, for example the kernel density estimate using the Gaussian kernel. Thus we have an estimate of the entropy index as follows.

$$\widehat{I}(a) = \int \log \widehat{p}(z)p(z)dz = E\left[\log \widehat{p}(z)\right] = \frac{1}{N}\sum_{i=1}^{N} \log \widehat{p}(z_i) = \frac{1}{N}\sum_{i=1}^{N} \log \widehat{p}(a^T x_i) \quad (6.38)$$

The entropy index $\widehat{I}(a)$ has to be optimized over the d-dimensional vector a subject to the constraint that $\|a\| = 1$. The optimization function will require the gradient of the objective function. For the index defined above the gradient can be written as

$$\frac{d}{da}[\widehat{I}(a)] = \frac{1}{N}\sum_{i=1}^{N} \frac{\widehat{p'}(a^T x_i)}{\widehat{p}(a^T x_i)} x_i. \quad (6.39)$$

For the PP the computational burden is greatly reduced if we use the proposed fast method. The computational burden is reduced in the following three instances.

1. Computation of the kernel density estimate (i.e. use the fast method with $r = 0$).

2. Estimation of the optimal bandwidth.

3. Computation of the first derivative of the kernel density estimate, which is required in the optimization procedure.

Fig. 6.3 shows an example of the application of the PP algorithm on an image segmentation problem. Fig. 6.3(a) shows the original 48 $\times$ 60 image of the hand with a ring against a background. Perceptually the image has three distinct regions,

the hand, the ring, and the background. Each pixel is represented as a point in a three dimensional RGB space. Fig. 6.3(b) shows the the presence of three clusters in the RGB space. We ran the PP algorithm on this space. Fig. 6.3(c) shows the KDE of the points projected on the most interesting direction. This direction is clearly able to distinguish the three clusters. Fig. 6.3(d) shows the segmentation where each pixel is assigned to the mode nearest to it.

The PP procedure was coded in MATLAB with the core computational task of computing the density derivative written in C++ with MATLAB wrappers. We used the MATLAB non-linear least squares routine *lsqnonlin* to perform the optimization. The tolerance parameter for the optimization procedure was set to 10^{-6}. The optimal bandwidth and the kernel density estimate were computed approximately. The accuracy parameter was set to $\epsilon = 10^{-3}$. The entire procedure took 15 minutes while that using the direct method takes around 7.5 hours.

6.9 Related work

Most of the past work has focussed on making KDE computationally tractable. There is no previous work specifically dealing with the computational complexity of bandwidth estimation.

The computational cost of evaluating the KDE (Eq. 6.5) at N points is $O(N^2)$. If the source points are on an evenly spaced grid then we can evaluate the sum at an evenly spaced grid exactly in $O(N \log N)$ using the fast Fourier transform (FFT). One of the earliest methods, especially proposed for univariate fast kernel density

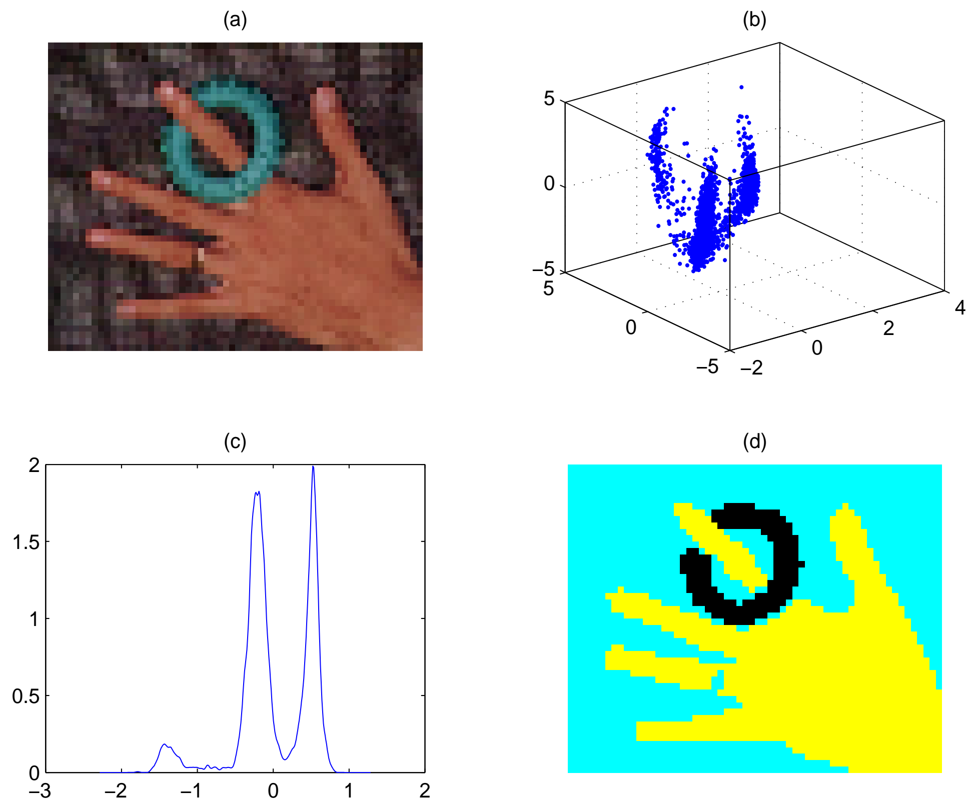


Figure 6.3: (a) The original image. (b) The centered and scaled RGB space. Each pixel in the image is a point in the RGB space. (c) KDE of the projection of the pixels on the most interesting direction found by projection pursuit. (d) The assignment of the pixels to the three modes in the KDE.

estimation was based on this idea [74]. For irregularly spaced data, the space is divided into boxes, and the data is assigned to the closest neighboring grid points to obtain grid counts. The KDE is also evaluated at regular grid points. For target points not lying on the the grid the value is obtained by doing some sort of interpolation based on the values at the neighboring grid points. As a result there is no guaranteed error bound for such kind of methods. The Fast Gauss Transform(FGT) [29] is an approximation algorithm that reduces the computational complexity to $O(N)$, at the expense of reduced precision, which can be specified. The constant depends on the desired precision, dimensionality of the problem, and the bandwidth. The improved fast Gauss transform(IFGT) is a similar algorithm based on a different factorization and data structures. It is suitable for higher dimensional problems and provides comparable performance in lower dimensions [96]. Another class of methods for such problems are dual-tree methods [26] which are based on space partitioning trees for both the source and target points. Using the tree data structure distance bounds between nodes can be computed. An advantage of the dual-tree methods is that they work for all common kernel choices, not necessarily Gaussian. However the series based methods give better speedups.

All the above methods are designed to specifically accelerate the KDE. The main contribution of this paper is to accelerate the kernel *density derivative* estimate with an emphasis to solve the optimal bandwidth problem. The case of KDE arises as a special case of $r = 0$, i.e., the zero order density derivative. While it is suggested in [29] that the FGT can also be used to accelerate the sum of polynomial times Gaussian; the specific details and error bounds are not provided. The FGT uses

the Hermite and Taylor series for factorizing the Gaussian. Our proposed method in comparison to the FGT uses a completely different single factorization.

6.10 Conclusions

We proposed an fast $\epsilon - exact$ algorithm for kernel density derivative estimation which reduced the computational complexity from $O(N^2)$ to $O(N)$. We demonstrated the speedup achieved for optimal bandwidth estimation both on simulated as well as real data. As an example we demonstrated how to potentially speedup the projection pursuit algorithm. We focussed on the univariate case in the current paper since the bandwidth selection procedures for the univariate case are pretty mature. Bandwidth selection for the multivariate case is a field of very active research [86]. Our future work would include the relatively straightforward but more involved extension of the current procedure to handle higher dimensions. As pointed out earlier many applications other than bandwidth estimation require derivative estimates. We hope that our fast computation scheme should benefit all the related applications.

Chapter 7

Gaussian process regression

Gaussian processes allow the treatment of non-linear non-parametric regression problems in a Bayesian framework. However the computational cost of training such a model with N examples scales as $\mathcal{O}(N^3)$. Iterative methods for the solution of linear systems can bring this cost down to $\mathcal{O}(N^2)$, which is still prohibitive for large data sets. We consider the use of ϵ-exact matrix-vector product algorithms to reduce the computational complexity to $\mathcal{O}(N)$. Using the theory of inexact Krylov subspace methods we show how to choose ϵ to guarantee the convergence of the iterative methods. We test our ideas using the improved fast Gauss transform. We demonstrate the speedup achieved on large data sets. For prediction of the mean the computational complexity is reduced from $\mathcal{O}(N)$ to $\mathcal{O}(1)$. Our experiments indicated that for low dimensional data ($d \leq 8$) the proposed method gives substantial speedups [62].

7.1 Introduction

The Gaussian process (GP) is a popular method for Bayesian non-linear non-parametric regression. Unfortunately its non-parametric nature causes computational problems for large data sets, due to an unfavorable $\mathcal{O}(N^3)$ time and $\mathcal{O}(N^2)$ memory scaling for training. While the use of iterative methods, as first suggested by [46], can reduce the cost to $\mathcal{O}(kN^2)$ where k are the number of iterations, this is still too large. An important subfield of work in GP has attempted to bring this scaling down to $\mathcal{O}\left(m^2N\right)$ by making sparse approximations of size m to the full GP where $m \ll N$ [91, 77, 19, 44, 14, 82, 81, 79]. Most of these methods are based on using a representative subset of the training examples of size m. Different schemes specify different strategies to effectively choose the subset. A good review can be found in Chapter 8 of [58] or [56] for a more recent survey. A recent work [79] considers choosing m datapoints not constrained to be a subset of the data. While these methods often work quite well, there is no guarantee on the quality of the GP that results from the sparse approximation.

A GP is completely specified by its mean and covariance functions. Different forms of the covariance function gives us the flexibility to model different kinds of generative processes. One of the most popular covariance function used is the negative squared exponential (Gaussian). In this paper we explore an alternative class of methods that seek to achieve a speed-up for GP regression by computing an ϵ-exact approximation to the matrix-vector product used in the conjugate gradient method. Unlike methods which rely on choosing a subset of the dataset we use all the avail-

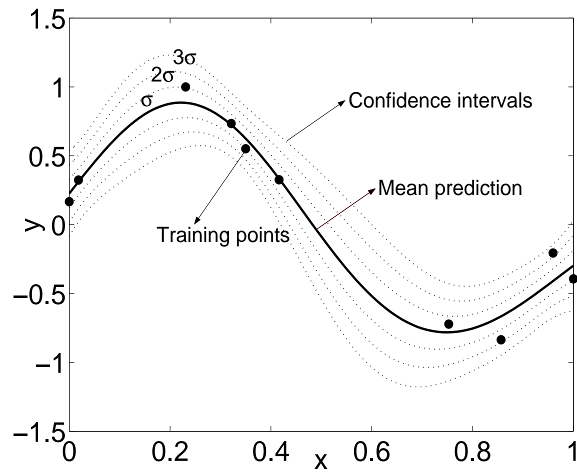
able points and still achieve $\mathcal{O}(N)$ complexity. There are at least three methods proposed to accelerate the matrix-vector product using approximation ideas: the dual-tree method [26], the fast Gauss transform (FGT) [29], and the improved fast Gauss transform (IFGT) [95], which have their own areas of applicability and performance characteristics. These methods claim to provide the matrix-vector product with a guaranteed accuracy ϵ, and achieve $\mathcal{O}(N \log N)$ or $\mathcal{O}(N)$ performance at fixed ϵ in both time and memory.

An important question when using these methods is the influence of the approximate matrix-vector product on the convergence of the iterative method. Obviously these methods converge at machine precision. However, the accuracy necessary to guarantee convergence must be studied. Generally previous papers [95, 73] choose ϵ to a convenient small value such as 10^{-3} or 10^{-6} based on the application. We use a more theoretical approach and base our results on the theory of inexact Krylov subspace methods. We show that the *matrix-vector product may be performed in an increasingly inexact manner* as the iteration progresses and still allow convergence.

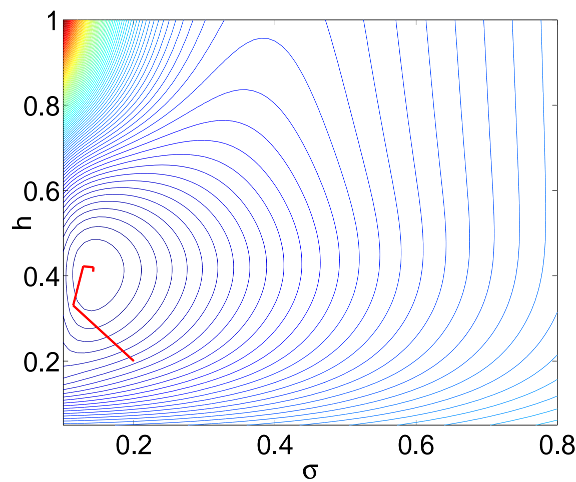
7.2 Gaussian process model

While Gaussian processes are covered well elsewhere (e.g. see [58]), both to establish notation and for completeness we provide a brief introduction here.

The simplest most often used model for regression [90] is $y = f(x) + \varepsilon$, where $f(x)$ is a zero-mean Gaussian process with covariance function $K(x, x') : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ and ε is independent zero-mean normally distributed noise with variance σ^2, i.e.,



(a)



(b)

Figure 7.1: (a) The mean prediction and the error bars obtained when a Gaussian process was used to model the data shown by the black points. A squared exponential covariance function was used. Note that the error bars increase when there is sparse data. The hyperparameters h and σ we chosen by minimizing the negative log-likelihood of the training data the contours of which are shown in (b).

$\mathcal{N}(0, \sigma^2)$. Therefore the observation process $y(x)$ is a zero-mean Gaussian process with covariance function $K(x, x') + \sigma^2 \delta(x, x')$.

Given training data $\mathcal{D} = \{x_i, y_i\}_{i=1}^N$ the $N \times N$ covariance matrix $\mathbf{K}$ is defined as $[\mathbf{K}]_{ij} = K(x_i, x_j)$. If we define the vector $\mathbf{y} = [y_1, \ldots, y_N]^T$ then $\mathbf{y}$ is a zero-mean multivariate Gaussian with covariance matrix $\mathbf{K} + \sigma^2\mathbf{I}$. Given the training data $\mathcal{D}$ and a new input x_* our task is to compute the posterior $p(f_*|x_*, \mathcal{D})$. Observing that the joint density $p(f_*, \mathbf{y})$ is a multivariate Gaussian, the posterior density $p(f_*|x_*, \mathcal{D})$ can be shown to be [58]

$$p(f_*|x_*, \mathcal{D}) \sim \mathcal{N}\left(\mathbf{k}(x_*)^T(\mathbf{K} + \sigma^2\mathbf{I})^{-1}\mathbf{y},\ K(x_*, x_*) - \mathbf{k}(x_*)^T(\mathbf{K} + \sigma^2\mathbf{I})^{-1}\mathbf{k}(x_*)\right), \tag{7.1}$$

where $\mathbf{k}(x_*) = [K(x_*, x_1), \ldots, K(x_*, x_N)]^T$.

If we define $\xi = (\mathbf{K} + \sigma^2\mathbf{I})^{-1}\mathbf{y}$, then the mean prediction and the variance associated with it are

$$\mathbb{E}[f_*] = \mathbf{k}(x_*)^T\xi, \quad \text{and} \tag{7.2}$$

$$\mathbb{V}\text{ar}[f_*] = K(x_*, x_*) - \mathbf{k}(x_*)^T(\mathbf{K} + \sigma^2\mathbf{I})^{-1}\mathbf{k}(x_*). \tag{7.3}$$

The covariance function has to be chosen to reflect the prior information about the problem. For high-dimensional problems, in the absence of any prior knowledge, the negative squared exponential (Gaussian) is the most widely used covariance function, and is the one that we use in this paper.

$$K(x, x') = \sigma_f^2 \exp\left(-\sum_{k=1}^{d} \frac{(x_k - x'_k)^2}{h_k^2}\right). \tag{7.4}$$

The $d + 2$ parameters ($[h_1, \ldots, h_d, \sigma_f, \sigma]$) are referred to as the hyperparameters.

7.3 Conjugate Gradient

Given the hyperparameters, the *training phase* consists of the evaluation of the vector

$$\xi = (\mathbf{K} + \sigma^2\mathbf{I})^{-1}\mathbf{y} \tag{7.5}$$

which needs the inversion of an $N \times N$ matrix $\mathbf{K} + \sigma^2\mathbf{I}$. Direct computation of the inverse of the symmetric matrix (using Cholesky decomposition) requires $\mathcal{O}(N^3)$ operations and $\mathcal{O}(N^2)$ storage, which is impractical even for problems of moderate size (typically a few thousands).

For larger systems it is more efficient to solve the system

$$\widetilde{\mathbf{K}}\xi = \mathbf{y} \qquad \text{where,} \quad \widetilde{\mathbf{K}} = \mathbf{K} + \sigma^2\mathbf{I} \tag{7.6}$$

using iterative methods, provided the method converges quickly. Modern iterative Krylov subspace methods show good convergence properties, especially when preconditioned [43]. Since $\widetilde{\mathbf{K}}$ is *symmetric and positive definite* we can use the well known *conjugate-gradient* (CG) method [36] to iteratively solve Eq. (7.6) A good exposition of this method can be found in Ch. 2 of [43]. The idea of using conjugate gradient for GP was first suggested by [46].

The iterative method generates a sequence of approximate solutions ξ_k at each step, which converge to the true solution ξ. One of the sharpest known convergence results for the iterates is given by

$$\frac{\|\xi - \xi_k\|_{\widetilde{\mathbf{K}}}}{\|\xi - \xi_0\|_{\widetilde{\mathbf{K}}}} \leq 2\left[\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right]^{2k}, \quad \|w\|_{\widetilde{\mathbf{K}}} = w^T\widetilde{\mathbf{K}}w \tag{7.7}$$

where the $\widetilde{\mathbf{K}}$-*norm* of any vector w is defined as above [43]. The constant $\kappa =$

$\lambda_{max}/\lambda_{min}$, the ratio of the largest to the smallest eigenvalues is called the *spectral condition number* of the matrix $\widetilde{\mathbf{K}}$. Since $\kappa \in (1, \infty)$, Equation 7.7 implies that if κ is close to one, the iterates will converge very quickly.

Given a tolerance $0 < \eta < 1$ a practical CG scheme iterates till it computes a vector ξ_k such that the ratio of the current residual $\|\mathbf{y} - \widetilde{\mathbf{K}}\xi_k\|_2$ to the initial residual is below the tolerance.

$$\frac{\|\mathbf{y} - \widetilde{\mathbf{K}}\xi_k\|_2}{\|\mathbf{y} - \widetilde{\mathbf{K}}\xi_0\|_2} \leq \eta. \tag{7.8}$$

Most implementations start the iteration at $\xi_0 = \mathbf{0}$, though a better guess can be used if available. The relative residual in the Euclidean norm is related to the relative error in the $\widetilde{\mathbf{K}}$-norm as [43]

$$\frac{\|\mathbf{y} - \widetilde{\mathbf{K}}\xi_k\|_2}{\|\mathbf{y} - \widetilde{\mathbf{K}}\xi_0\|_2} \leq \sqrt{\kappa}\frac{\|\xi - \xi_k\|_{\widetilde{\mathbf{K}}}}{\|\xi - \xi_0\|_{\widetilde{\mathbf{K}}}} \leq 2\sqrt{\kappa}\left[\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right]^{2k}. \tag{7.9}$$

This implies that for a given η the number of iterations required is

$$k \geq \ln\left[\frac{2\sqrt{\kappa}}{\eta}\right] / 2\ln\left[\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right]. \tag{7.10}$$

Sometimes the estimate (7.10) can be very pessimistic. Even if the condition number is large, the convergence is fast if the eigenvalues are clustered in a few small intervals [43]. In the examples we consider later convergence was achieved relatively quickly. If convergence is slow we must consider preconditioning, which is a topic outside the scope of the present paper.

The actual implementation of the CG method requires *one* $\mathcal{O}(dN^2)$ *matrix-vector multiplication* and $5N$ flops per iteration. Four vectors of length N are required for storage. The storage is $\mathcal{O}(N)$ since the matrix-vector multiplication can

use elements computed on the fly and not storing the entire matrix. Empirically the number of iterations required is generally small compared to N leading to a computational cost of $\mathcal{O}(kdN^2)$. It should be noted that the $\mathcal{O}(N)$ space comes at a time trade-off. If the matrix is cached (*i.e.* $\mathcal{O}(N^2)$ memory) then the computational cost is $\mathcal{O}(dN^2 + kN^2)$.

7.4 Fast matrix-vector products

The quadratic complexity is still too high for large datasets. The core computational step in each CG iteration involves the multiplication of the matrix $\mathbf{K}$ with some vector, say $\mathbf{q}$. The jth element of the matrix-vector product $\mathbf{Kq}$ can be written as $(\mathbf{Kq})_j = \sum_{i=1}^{N} q_i k(x_i, x_j)$.

In general for each *target point* $\{t_j \in \mathbb{R}^d\}_{j=1}^{M}$ (which in our case are the same as the *source points* x_i) this can be written as

$$G(t_j) = \sum_{i=1}^{N} q_i k(x_i, t_j). \tag{7.11}$$

The computational complexity to evaluate (7.11) at M target points is $\mathcal{O}(MN)$. For the Gaussian kernel various approximation algorithms have been proposed to compute the above sum in $\mathcal{O}(M+N)$ time. These algorithms compute the sum to any arbitrary ϵ precision. Broadly there are two kinds of methods–the series based methods and data structure based methods.

Series based methods: These methods are inspired by the fast multipole methods (FMM) which were originally developed for the fast summation of the potential fields generated by a large number of sources, such as those arising in

gravitational potential problems [28]. The fast Gauss transform (FGT) is a special case where FMM ideas were used for the Gaussian potential [29]. The improved fast Gauss transform (IFGT) is a similar algorithm based on a single different factorization and data structures. It is suitable for higher dimensional problems and provides comparable performance in lower dimensions [95].

Data structure based methods: Another class of methods proposed are the dual-tree methods [26]. These methods rely on space partitioning trees like kd-trees and ball trees and not on series expansions.

7.5 The accuracy ϵ, necessary

Obviously the accuracy, ϵ, that minimizes work while achieving the best performance must be chosen. However, determining this quantity in a principled way is often difficult. Most previous methods choose ϵ to a convenient small value such as 10^{-3} or 10^{-6} based on the application and *a posteriori* analysis. Indeed, one can in principle adaptively vary ϵ as the iteration proceeds. We were however able to use some recent results from linear algebra [75] and analyze the effect of the choice of ϵ on the CG method.

The conjugate gradient method is a *Krylov subspace method* adapted for a symmetric positive definite matrix. Krylov subspace methods at the kth iteration compute an approximation to the solution of any linear system $Ax = b$ by minimizing some measure of error over the affine space $x_0 + \mathcal{K}_k$, where x_0 is the initial iterate and the kth Krylov subspace is $\mathcal{K}_k = \text{span}(r_0, Ar_0, A^2r_0, \ldots, A^{k-1}r_0)$. The residual

at the kth iterate is $r_k = b - Ax_k$.

A general framework for understanding the effect of approximate matrix-vector products on Krylov subspace methods for the solution of symmetric and nonsymmetric linear systems of equations is given in [75]. The paper considers the case where at the kth iteration instead of the exact matrix-vector multiplication Av_k, the product

$$\mathcal{A}v_k = (A + E_k)v_k \tag{7.12}$$

is computed, where E_k is an error matrix which may change as the iteration proceeds. A nice result in the paper shows how large $\|E_k\|$ can be at each step while still achieving convergence with the desired tolerance. Let $r_k = \|Ax_k - b\|$ be the residual at the end of the kth iteration. Let $\tilde{r}_k$ be the corresponding residual when an approximate matrix-vector product is used. If at every iteration

$$\|E_k\| \leq l_m \frac{1}{\|\tilde{r}_{k-1}\|}\delta, \tag{7.13}$$

then at the end of k iterations $\|\tilde{r}_k - r_k\| \leq \delta$ [75]. The term l_m in general is unavailable since it depends on knowing the spectrum of the matrix. However our empirical results and also some experiments in [75] suggest that $l_m = 1$ seems to be a reasonable value. This shows that the matrix-vector product may be performed *in an increasingly inexact manner as the iteration progresses and still allow convergence to the solution.*

We will use the following notion of ϵ-exact approximation. Given any $\epsilon > 0$, $\hat{G}(t_j)$ is an ϵ-exact approximation to $G(t_j)$ if the maximum absolute error relative

to the total weight $Q = \sum_{i=1}^{N} |q_i|$ is upper bounded by ϵ, i.e.,

$$\max_{t_j} \left[\frac{|\hat{G}(t_j) - G(t_j)|}{Q} \right] \leq \epsilon. \tag{7.14}$$

For our problem because of the ϵ-exact approximation criterion (Equation 7.14) every element in the approximation to the vector $\mathbf{Kq}$ is within $\pm Q\epsilon_k$ of the true value, where $Q = \sum_{i=1}^{N} |q_i|$ and ϵ_k is the error in the matrix vector product at the kth iteration. Hence the error matrix E_k is of the form

$$E_k = \epsilon_k \begin{pmatrix} \pm e_{11} & \dots & \pm e_{1N} \\ \vdots & \ddots & \vdots \\ \pm e_{N1} & \dots & \pm e_{NN} \end{pmatrix}, \tag{7.15}$$

where $e_{ij} = sign(q_j) \in (+1, -1)$. It should be noted that this matrix is an upper bound rather than the actual error matrix. It can be seen that $\|E_k\| = N\epsilon_k$. Hence Equation 7.13 suggests the following strategy to choose ϵ_k.

$$\epsilon_k \leq \frac{\delta}{N} \frac{\|\mathbf{y} - \widetilde{\mathbf{K}}\xi_0\|}{\|\tilde{r}_{k-1}\|}. \tag{7.16}$$

This guarantees that

$$\frac{\|\mathbf{y} - \widetilde{\mathbf{K}}\xi_k\|_2}{\|\mathbf{y} - \widetilde{\mathbf{K}}\xi_0\|_2} \leq \eta + \delta. \tag{7.17}$$

Figure 7.2 shows the ϵ_k selected at each iteration for a sample regression problem. As the iteration progresses the ϵ_k required increases.

7.6 Prediction

Once ξ is computed in the training phase, the mean prediction for any new x_* is given by $\mathbb{E}[f_*] = \mathbf{k}(x_*)^T\xi = \sum_{i=1}^{N} \xi_i k(x_i, x_*)$. Predicting at M points is again

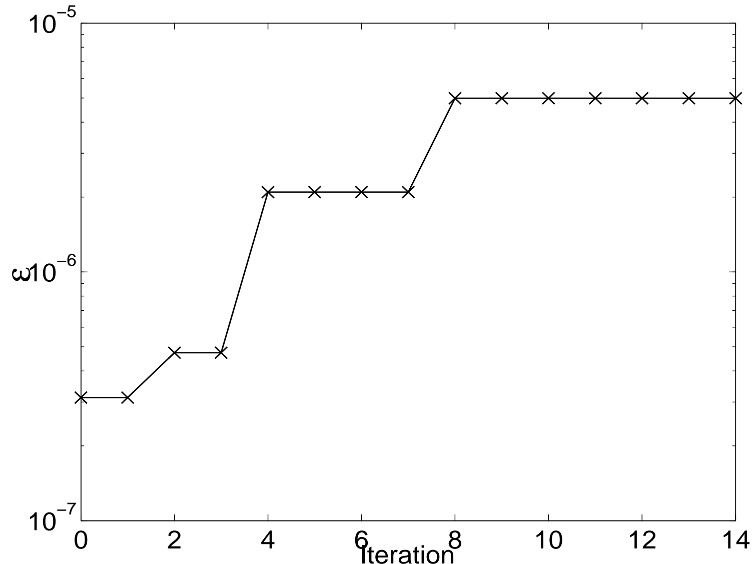


Figure 7.2: The error for the IFGT ϵ_k selected at each iteration for a sample 1D regression problem. The error tolerance for the CG was set to $\eta = 10^{-3}$ and $\delta = 10^{-3}$.

a matrix vector multiplication operation. Direct computation of $\mathbb{E}[f_*]$ at M test points due to the N training examples is $\mathcal{O}(NM)$. Using the fast matrix-vector product reduces the computational cost to $\mathcal{O}(N + M)$.

The variance for each prediction is given by $\mathbb{V}\text{ar}[f_*] = K(x_*, x_*) - \mathbf{k}(x_*)^T(\mathbf{K} + \sigma^2\mathbf{I})^{-1}\mathbf{k}(x_*)$. First we need to solve a linear system with $\mathbf{K} + \sigma^2\mathbf{I}$ during the training phase via some suitable decomposition. Once the decomposition is computed for each x the computation of uncertainty is $\mathcal{O}(N^2)$. For M points it is $\mathcal{O}(MN^2)$. Using the conjugate gradient method and the IFGT we can compute $\widetilde{\mathbf{K}}^{-1}\mathbf{k}(x_*)$ in $\mathcal{O}(kN)$ time. For M points we need $\mathcal{O}(kMN)$ time.

Table 7.1 compares the computational and space complexities for different stages of Gaussian process regression using different methods.

Table 7.1: *The dominant computational and space complexities.* We have N training points and M test points in d dimensions. k is the number of iterations required by the conjugate gradient procedure to converge to a specified tolerance. The memory requirements are for the case where the Gram matrix is constructed on the fly at each iteration. For the fast MVM procedure, the constant $\mathcal{D}$ grows with d depending on the type of fast MVM used.

	Direct Method		*Conjugate gradient*		*Conjugate gradient+Fast MVM*	
	Time	Space	Time	Space	Time	Space
Training	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$	$\mathcal{O}(kdN^2)$	$\mathcal{O}(dN)$	$\mathcal{O}(k\mathcal{D}N)$	$\mathcal{O}(dN+\mathcal{D})$
Prediction	$\mathcal{O}(dMN)$	$\mathcal{O}(dM+dN)$			$\mathcal{O}(\mathcal{D}M+\mathcal{D}N)$	$\mathcal{O}(dM+dN+\mathcal{D})$
Uncertainty	$\mathcal{O}(MN^2)$		$\mathcal{O}(kdMN^2)$	$\mathcal{O}(dM+dN)$	$\mathcal{O}(k\mathcal{D}MN)$	$\mathcal{O}(dM+dN+\mathcal{D})$

7.7 Experiemnts

Datasets: We use the following two datasets – ***robotarm*** [1][dataset size $N = 10,000$, dataset dimension $d = 2$] and ***abalone*** [2][$N = 4,177, \quad d = 7$]. The datasets are chosen to be representative of low and medium dimensions respectively. These are also known to be highly non-linear regression problems and widely used to benchmark regression algorithms.

Evaluation Procedure: For each dataset 90% of the examples were used for training and the remaining 10% were used for testing. The results are shown for a ten-fold cross validation experiment. The inputs are linearly re-scaled to have zero mean and unit variance on the training set. The outputs are centered to have zero mean on the training set. The mean squared error (MSE) is defined as the squared error between the mean prediction and the actual value averaged over the test set. Since the MSE is sensitive to the overall scale of the target values we normalize it by the variance of the targets of the test cases to obtain the *standardized mean squared error* (SMSE) [58]. This causes the trivial method of guessing the mean of the training targets to have a SMSE of approximately 1. For all the experiments we used the squared exponential covariance function

[1] A synthetic 2-d nonlinear robot arm mapping problem [47]. The data is generated according to $f(x_1, x_2) = 2.0\cos(x_1) + 1.3\cos(x_1 + x_2)$. The value of x_1 is chosen randomly in $[-1.932, -0.453]$ and x_2 is chosen randomly in $[0.534, 3.142]$ as in [58]. The target values are obtained by adding Gaussian noise of variance 0.1 to $f(x_1, x_2)$.

[2] The task is to predict the age of abalone (number of rings) from physical measurements. Downloaded from `http://www.liacc.up.pt/~ltorgo/Regression/DataSets.html`

(Equation 7.4). The $d+2$ hyperparameters ($[h_1, \ldots, h_d, \sigma_f, \sigma]$) were selected by optimizing the marginal likelihood on the subset using the direct method (automatic relevance determination (ARD) [90]) [3]. For all methods the same hyperparameters were used. For larger subsets where we cannot find the hyperparameters directly we use the one computed from the largest possible subset.

Results: Figure 7.3 shows the total training time, the SMSE, and the total prediction time for the two datasets as a function of the number of datapoints. For each fold a subset of the training data of size m was selected at random. The process was repeated 10 times. m was progressively increased to get a learning curve. All the experiments were run on a 1.83 GHz processor with 1GB of RAM. We show the scaling behavior for the following four methods.

1. ***Subset of datapoints*** (SD) This is simply the direct implementation with a subset of the training data. The subset is chosen randomly. The training and prediction time scale as $\mathcal{O}(m^3)$ and $\mathcal{O}(mM)$ respectively. M is the total number of test points.

2. ***Subset of regressors/projected process*** (SR and PP) (See Chapter 8 in [58] for a description of these methods.) The training and prediction time scale as $\mathcal{O}(m^2N)$ and $\mathcal{O}(mM)$ respectively. N is the total number of training points. The SR and PP methods have the same predictive mean. The recent paper [79] also has the same computational complexity [4].

[3] Code downloaded from `http://www.gaussianprocess.org/gpml/code/matlab/doc/`.

[4] Also it is expected to be much more expensive because of the optimization procedure to find the location of the pseudo-inputs.

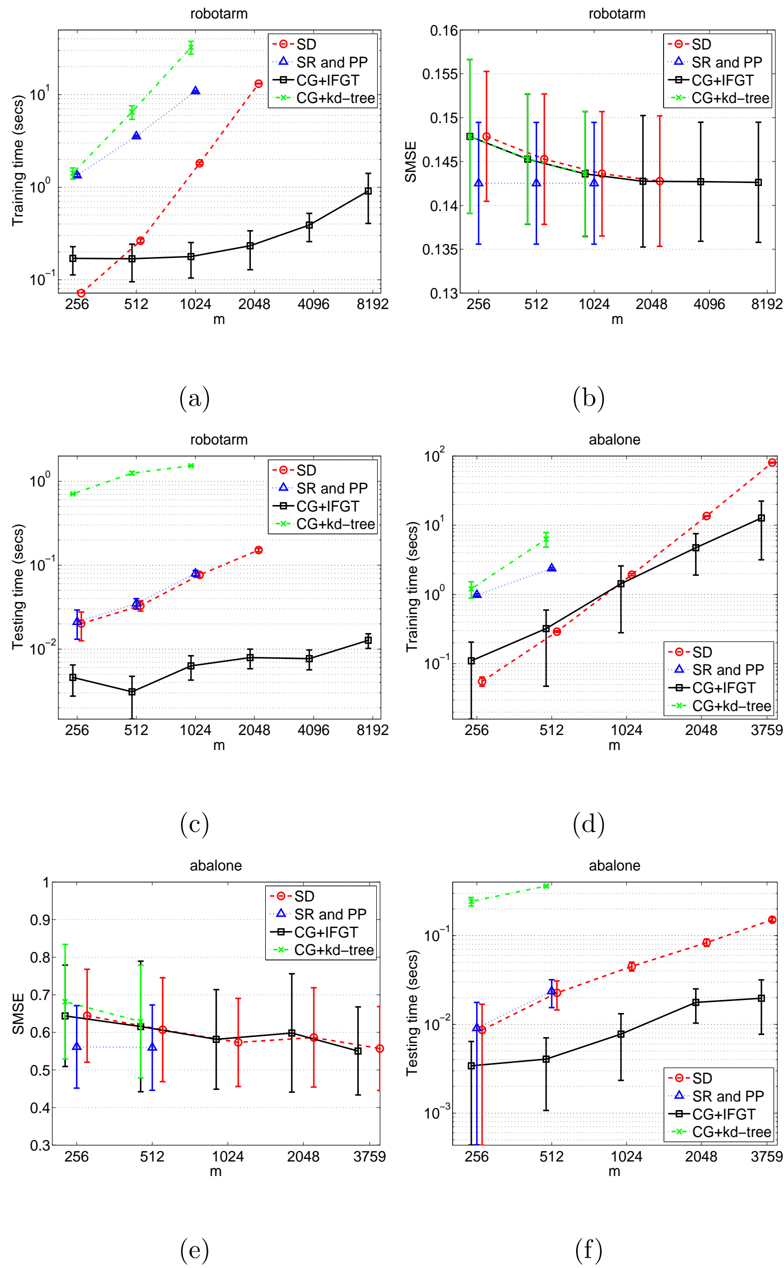


Figure 7.3: (a) The total training time, (b) the SMSE, and (c) the testing time as a function of m for the robotarm dataset. The errorbars show one standard deviation over a 10-fold experiment. The results are slightly displaced w.r.t. the horizontal axis for clarity. The lower panel show the same for the abalone dataset.

3. ***Proposed method with IFGT*** (CG+IFGT) The training and prediction time scale as $\mathcal{O}(km)$ and $\mathcal{O}(m+M)$ respectively. The tolerance for the conjugate gradient procedure η was set to 10^{-3} and the δ in Equation 7.17 for IFGT was set to 10^{-3}. The accuracy for testing using IFGT was set to $\epsilon = 10^{-6}$.

4. ***Proposed method with kd-tree***(CG+kd-tree) Same as above but using kd-tree instead of the IFGT.

The following observations can be made:

- From Figure 7.3(a) it can be seen that as m increases the training time for the proposed method increases linearly in contrast to the quadratic increase for the SD method. The SR and PP methods have small training times only for small m.

- As m increases the general trend for all methods is that SMSE decreases (see Figure 7.3(b)).

- It is not surprising that SR and PP show the least SMSE. This is because SR and PP use all the datapoints while retaining m of them as the active set. However the proposed method can still catch up with the SMSE of SR and PP and still have a significantly lower running time.

- Regarding the testing time the proposed method shows significant speedups.

- As the dimension of the problem increases the cutoff point, i.e., N at which the proposed fast method is better than the direct method increases.

- At the hyperparameters chosen, the dual-tree algorithms ended up taking larger time than the direct method probably because of the time taken to build up the kd-trees.

For large dimensional data the fast algorithms like IFGT and dual-tree methods do not scale well. We were unable to get good speedups for high dimensional datasets like SARCOS [5] (a 21 dimensional robot arm dataset) using the IFGT. However either the subset of data or PP/SR methods can be used with a higher dimensional data set such as SARCOS.

7.8 Implicit Surface fitting

Recently implicit models for surface representation are gaining popularity [9, 69, 85]. An implicit representation describes the surface S as the set of all points where a certain smooth function, $f : \mathbb{R}^d \rightarrow \mathbb{R}$ vanishes, *i.e.*, $\mathrm{S} = f^{-1}(0) = \{x \in \mathbb{R}^d \text{ such that } f(x) = 0\}$. Once we have a representation f it can be evaluated on a grid in $\mathbb{R}^d$ and an explicit representation can be formed as a mesh of polygons for visualization purposes–often referred to as isosurface extraction.

Given a set of N points $\{x_i \in \mathbb{R}^d\}_{i=1}^N$ ($d = 3$ for surface fitting) lying on a smooth manifold S we have to find a function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ such that $f(x_i) = 0, \text{ for } i = 1, \ldots, N$, and it smoothly interpolates for any other $x \in \mathbb{R}^d$. In order to avoid the trivial solution $f(x) = 0$ we need to add additional constraints, *i.e.*, points where the function f is not zero. Such additional points are referred to as

[5] http://www.gaussianprocess.org/gpml/data/

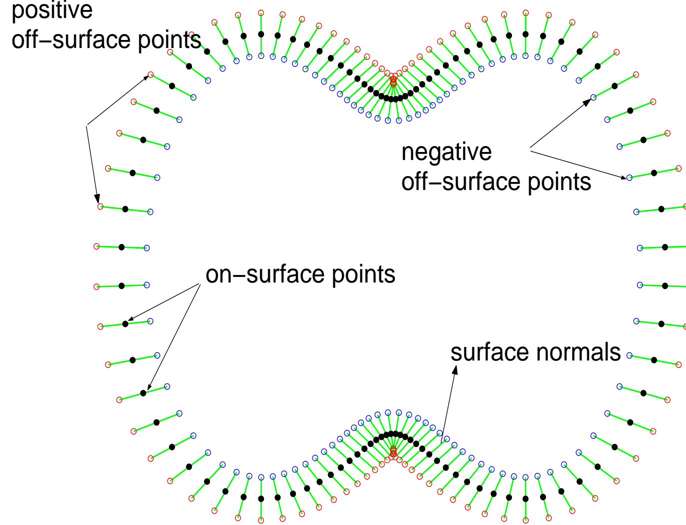


Figure 7.4: Appending the positive and the negative off-surface points along the surface normals at each point.

the *off-surface* points. So the formulation now is to find a function f such that $f(x_i) = 0$, for $i = 1, \ldots, N$ and $f(x_i) = d_i \neq 0$, for $i = N+1, \ldots$. For generating the off-surface points we use the following scheme [9]. We append each data point x_i with two off-surface points x_i^+ and x_i^-, one on each side of the surface as shown in Figure 7.4. The off-surface points are generated by projecting along the surface normals at each point. It should be ensured that the surface normals are consistently oriented. We use the signed-distance function for the off-surface points. The value of the function is chosen to be the distance to the closest on-surface point. Points outside the surface are assigned a positive value while those inside the surface are given a negative value. We ensure that the off-surface points do not intersect the underlying surface using the normal length validation scheme described in [9].

We have used Gaussian process regression to learn this function from the

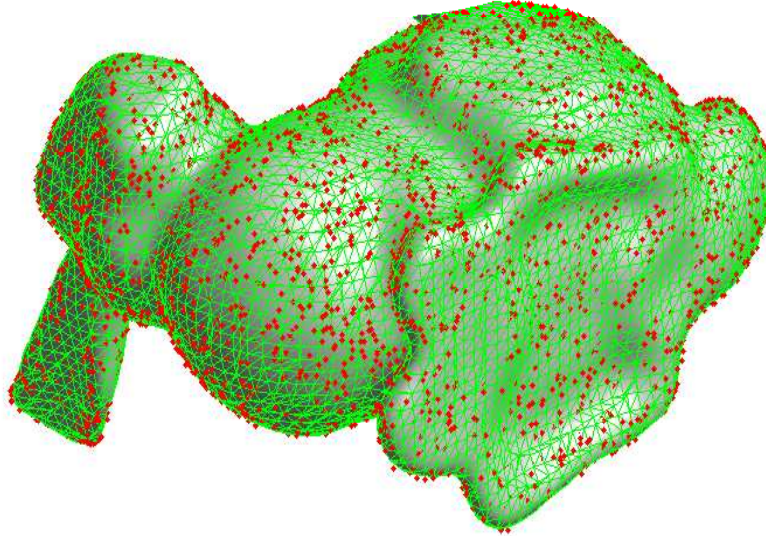


Figure 7.5: The isosurface extracted using the function learnt by Gaussian Process regression. The point cloud data is also shown. It took 6 minutes to learn this implicit surface form $10,000$ surface points.

point cloud data. One of the major bottlenecks for most implicit surface methods is their prohibitive computational complexity, and this applies to Gaussian process regression as well, the computational complexity of which scales as $\mathcal{O}(N^3)$. Using the propose method we were able to handle large data-sets. Since surface fitting in done in $d = 3$ IFGT gave good speedups. Figure 7.5 shows the fitted surface for the *bunny* data. It took 6 minutes to fit the model using 10,000 surface points. Note that the actual number of points used for Gaussian process regression is 30,000 due to the off-surface points. We were unable to run the direct method on such large datasets. Unlike regression for surface fitting we would like to use all the available data to get accurate surface reconstruction. The IFGT was also used for isosurface extraction.

7.9 Discussion/Further issues

We have demonstrated that the approximate fast matrix vector products achieved by ϵ-exact methods such as the improved fast Gauss transform can achieve a fast solution of the Gaussian process regression. The following are the contributions of this paper:

(1) We show that the training time for GP regression is reduced to linear $\mathcal{O}(N)$ by using the conjugate-gradient method coupled with the IFGT. The prediction time per test input is reduced to $\mathcal{O}(1)$.

(2) Using results from the theory of inexact Krylov subspace methods we show that the matrix-vector product may be performed in an increasingly inexact manner as the iteration progresses and still allow convergence to the correct solution.

(3) Our experiments indicated that for low dimensional data ($d \leq 8$) the proposed method gives substantia speedups.

The idea of speeding up matrix-vector multiplication for Gaussian process regression was first explored in [73]–who use kd-trees to speed up the matrix-vector multiplication. The main contribution of this paper is a strategy to choose ϵ while using such methods.

While the scope of this paper is to speed up the original GPR it should be noted that methods which use a subset of the data [91, 77, 19, 44, 14, 82, 81] can also be further speeded up using these algorithms. This is because even these methods require matrix-vector products to be taken with a smaller subset of the data.

One drawback of the IFGT is that it is specific to the Gaussian kernel. For

other covariance functions, like the Matern class of kernels–fast algorithms can be developed. The results presented in this paper, regarding the choice of ϵ should hold independent of the covariance function used.

It would also be interesting to explore whether the techniques presented here can be used to speedup classification [92] using a Gaussian process model.

Chapter 8

Large scale preference learning

Largely motivated by applications in search engines, information retrieval, and collaborative filtering, ranking has recently received significant attention in the statistical machine learning community. In a typical formulation, we compare two instances and determine which one is better or preferred. Based on this, a set of instances can be ranked according to the desired preference relation. Many ranking algorithms have been proposed in the literature. Most of them learn a ranking function from pairwise relations. However they are computationally expensive to train due to the quadratic scaling in the number of pairwise constraints, thus seriously restricting the use of ranking formulations to large datasets. In this Chapter I will describe a new algorithm that runs in linear time. While our algorithm also uses pairwise comparisons the runtime is still linear. This is made possible by fast approximate summation of erfc functions described in Chapter 4. Experiments on public benchmarks for ordinal regression and collaborative filtering show that the proposed algorithm is as accurate as the best available methods in terms of ranking accuracy, when trained on the same data, and is several orders of magnitude faster. [63, 66, 67]

8.1 Introduction

The problem of *ranking* has recently received significant attention in the statistical machine learning and information retrieval communities. In a typical ranking formulation, we compare two instances and determine which one is *better* or *preferred*. Based on this, a set of instances can be ranked according to a desired *preference relation*. The study of ranking has largely been motivated by applications in search engines, information retrieval, collaborative filtering, and recommender systems. For example in search engines, rather than returning a document as relevant or not (classification), the ranking formulation allows one to sort the documents in the order of their relevance.

8.1.1 Preference relation and ranking function

Consider an instance space $\mathcal{X}$. For any $(x, y) \in \mathcal{X} \times \mathcal{X}$ we interpret the *preference relation* $x \succeq y$ as '*x is at least as good as y*'. We say that '*x is indifferent to y*' ($x \sim y$) if $x \succeq y$ and $y \succeq x$. For *learning a ranking* we are provided with a set of pairwise preferences, based on which we have to learn a preference relation. In general, an ordered list of instances can always be decomposed down to a set of pairwise preferences.

One way of describing preference relations is by means of a ranking function. A function $f : \mathcal{X} \to \mathbb{R}$ is a *ranking/scoring function* representing the preference relation $\succeq$ if

$$\forall x, y \in \mathcal{X}, \quad x \succeq y \Leftrightarrow f(x) \geq f(y). \tag{8.1}$$

The ranking function f provides a numerical score to the instances based on which the instances can be ordered. Note that the function f is not unique. For any strictly increasing function $g : \mathbb{R} \rightarrow \mathbb{R}$, $g(f(.))$ is a new ranking function representing the same preference relation. It may be noted that $x \sim y \Leftrightarrow f(x) = f(y)$.

The ranking function is similar to the *utility function* used in microeconomic theory [50, 33], where utility is a measure of the satisfaction gained by consuming commodities. A consequence of using a ranking function is that the learnt preference relation is *rational*. In economics a preference relation $\succeq$ is called rational if it satisfies the following two properties [50]:

- *Completeness:* For all $x, y \in \mathcal{X}$, we have that $x \succeq y$ or $y \succeq x$.
- *Transitivity:* For all $x, y, z \in \mathcal{X}$, if $x \succeq y$ and $y \succeq z$ then $x \succeq z$.

A preference relation can be represented by a ranking function only if it is rational: For all $x, y \in \mathcal{X}$ either $f(x) \geq f(y)$ or $f(y) \geq f(x)$. This proves the completeness property. For all $x, y, z \in \mathcal{X}$, $f(x) \geq f(y)$ and $f(y) \geq f(z)$, implies that $f(x) \geq f(z)$. Hence transitivity is satisfied.

A central tenet of microeconomic theory is that most of the human preferences can be assumed to be rational [50]. In the training data we may have preferences which do not obey transitivity; However, the learnt ranking function will correspond to a rational preference relation. With a slight abuse of terminology, for the rest of the chapter we shall simply treat the learning of a preference relation as a problem of learning a rational ranking function, f.

8.1.2 Problem statement

In the literature, the problem of learning a ranking function has been formalized in many ways. We adopt the most general formulation based on directed preference graphs [23, 16].

We are given training data $\mathcal{A}$, a directed preference graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ encoding the preference relations, and a function class $\mathcal{F}$ from which we choose our ranking function f.

- The training data $\mathcal{A} = \bigcup_{j=1}^{S}(\mathcal{A}^j = \{x_i^j \in \mathbb{R}^d\}_{i=1}^{m_j})$ contains S classes (sets). Each class $\mathcal{A}^j$ contains m_j samples and there are a total of $m = \sum_{j=1}^{S} m_j$ samples in $\mathcal{A}$.

- Each vertex of the directed order graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ corresponds to a class $\mathcal{A}^j$. The existence of a directed edge $\mathcal{E}_{ij}$ from $\mathcal{A}^i \to \mathcal{A}^j$ means that all training samples in $\mathcal{A}^j$ are *preferred* or *ranked higher* than any training sample in $\mathcal{A}^i$, *i.e.* , $\forall(x_k^i \in \mathcal{A}^i, x_l^j \in \mathcal{A}^j)$, $x_l^j \succeq x_k^i$ (See Figure 8.1).

The goal is to learn a ranking function $f : \mathbb{R}^d \to \mathbb{R}$ such that $f(x_l^j) \succeq f(x_k^i)$ for as many pairs as possible in the training data $\mathcal{A}$ and also to perform well on unseen examples. The output $f(x_k)$ can be sorted to obtain a rank ordering for a set of test samples $\{x_k \in \mathbb{R}^d\}$.

This general formulation gives us the flexibility to learn different kinds of preference relations by changing the preference graph. Figure 8.1 shows two different ways to encode the preferences for a ranking problem with 4 classes. The first one containing all possible relations is called the full preference graph.

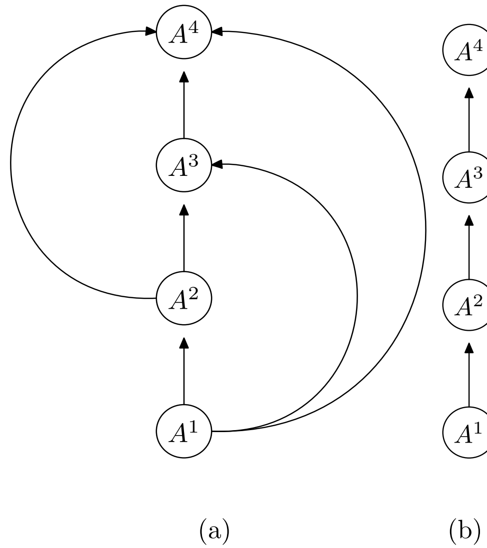


Figure 8.1: (a) A full preference graph and (b) chain preference graph for a ranking problem with 4 classes.

While a ranking function can be obtained by learning classifiers or ordinal regressors, it is more advantageous to learn the ranking function directly due to two reasons. First, in many scenarios it is more natural to obtain training data for pair-wise preference relations rather than the actual labels for individual samples. Second, the loss function used for measuring the accuracy of classification or ordinal regression—*e.g.* the 0-1 loss function—is computed for every sample individually, and then averaged over the training or the test set. In contrast, to asses the quality of the ranking for arbitrary preference graphs, we will use a generalized version of the *Wilcoxon-Mann-Whitney* (WMW) statistic [89, 48, 23] that is averaged over *pairs* of samples

8.1.3 Generalized Wilcoxon-Mann-Whitney statistic

The Wilcoxon-Mann-Whitney (WMW) statistic [89, 48] is frequently used to assess the performance of a classifier because of its equivalence to the area under the ROC (Receiver Operating Characteristics) curve (AUC). It is equal to the probability that a classifier assigns a higher value to the positive example than to the negative example, for a randomly drawn *pair of samples.* The generalized version of the WMW for the ranking problem is defined as follows [23]

$$\text{WMW}(f, \mathcal{A}, \mathcal{G}) = \frac{\sum_{\mathcal{E}_{ij}} \sum_{k=1}^{m_i} \sum_{l=1}^{m_j} \mathbf{1}_{f(x_l^j) \geq f(x_k^i)}}{\sum_{\mathcal{E}_{ij}} \sum_{k=1}^{m_i} \sum_{l=1}^{m_j} 1}, \tag{8.2}$$

$$\text{where} \quad \mathbf{1}_{a \geq b} = \begin{cases} 1 & \text{if } a \geq b \\ 0 & \text{otherwise} \end{cases} \tag{8.3}$$

The numerator counts the number of correct pairwise orderings. The denominator is the total number of pairwise preference relations available. The WMW is an estimate of $\Pr[f(x_1) \geq f(x_0)]$ for a randomly drawn pair of samples (x_1, x_0) such that $x_1 \succeq x_0$. This is a generalization of the area under the ROC curve (often used to evaluate bipartite rankings), to arbitrary preference graphs between many classes of samples. For a perfectly ranking function WMW=1, and for a completely random assignment WMW=0.5.

A slightly more general formulation can be found in [8, 16, 20], where each edge in the graph has an associated weight which indicates the strength of the preference relation. In such a case each term in the WMW must be suitably weighted.

While the WMW has been used widely to evaluate a learnt model, it has only recently been used as an objective function to learn the model. Since maximizing

the WMW is a discrete optimization problem most previous algorithms optimize a continuous relaxation instead. Even though the WMW itself can be computed with $\mathcal{O}(md + m \log m)$ computational effort, the previous algorithms often incurred $\mathcal{O}(m^2)$ effort in order to evaluate the relaxed version or its gradient. This seriously restricted the use of ranking formulations to large scale datasets. Because of this, most implementations could typically handle datasets containing only a few thousand samples.

8.1.4 Our proposed approach

We choose f to be a linear function and directly maximize the relaxed version of the WMW statistic using a conjugate gradient (CG) optimization procedure. The gradient computation scales as $\mathcal{O}(dm^2)$ which is computationally intractable for large datasets. Inspired by the fast multipole methods in computational physics [27], we develop a new algorithm that allows us to compute the gradient approximately to ϵ precision in $\mathcal{O}(dm)$ time. This enables the learning algorithm to scale well to large datasets.

8.1.5 Organization

The rest of the paper is structured as follows. In Section 8.2 we describe the previous work in ranking and place our method in context. The cost function which we optimize is described in Section 8.3. We also show that the cost function derived from a probabilistic framework can be considered as a lower bound on the WMW

(see Section 8.3.1). The computational complexity of the gradient computation is analysed in Section 8.4.2. In Section 8.5 we describe the fast summation of erfc functions which makes the learning algorithm scalable for large datasets. Experimental results are presented in Section 8.6.

8.2 Previous literature on learning ranking functions

Many ranking algorithms have been proposed in the literature. Most learn a ranking function from pairwise relations, and as a consequence are computationally expensive to train as the number of pairwise constraints is quadratic in the number of samples.

8.2.1 Methods based on pair-wise relations

The problem of learning rankings was first treated as a classification problem on pairs of objects by Herbrich et al [33] and subsequently used on a web page ranking task by Joachims [40]. The positive and negative examples are constructed from pairs of training examples–e.g., Herbrich et al [33] use the difference between the feature vectors of two training examples as a new feature vector for that pair. Algorithms similar to SVMs were used to learn the ranking function.

Burges et al. [8], proposed the RankNet which uses a neural network to model the underlying ranking function. Similar to our approach it uses gradient descent techniques to optimize a probabilistic cost function–the cross entropy. The neural net is trained on pairs of training examples using a modified version of backpropa-

gation algorithm.

Herbrich et al. [34] cast the ranking problem as an ordinal regression problem. The actual ranks are modeled as intervals on the real line. Hence rank boundaries play a critical role during training. The loss function depends on pairs of examples and their target ranks.

Several boosting based algorithms have been proposed for ranking. With collaborative filtering as an application Freund et al. [20] proposed the RankBoost algorithm for combining preferences. Dekel et al. [16] present a general framework for label ranking by means of preference graphs and graph decomposition procedure. A log-linear model is learnt using a boosting algorithm.

8.2.2 Fast approximate algorithms

The naive optimization strategy proposed in all the above algorithms suffer from the $\mathcal{O}(m^2)$ growth in the number of constraints. Approximation methods have recently been investigated. An approximate Expectation Propagation algorithm for Bayesian inference for Gaussian Processes was proposed in [88]. An efficient implementation of the RankBoost algorithm for two class problems was presented in [20]. A convex-hull based relaxation scheme was proposed in [23]. In a recent paper Yan and Hauptmann [94] proposed an approximate margin-based rank learning framework by bounding the pairwise risk function. This reduced the computational cost of computing the risk function from quadratic to linear.

8.2.3 Other approaches

A parallel body of literature has considered online algorithms and sequential update methods which find solutions in single passes through the data. PRank [12, 32] is an perceptron based online ranking algorithm which learns using one example at a time. RankProp [10] is a neural net ranking model which is trained on individual examples rather than pairs. However it is not known whether the algorithm converges. All gradient based learning methods can also be trained using stochastic gradient descent techniques.

8.2.4 WMW maximizing algorithms

Our proposed algorithm directly maximized the WMW. Previous algorithms which explicitly try to maximize the WMW come in two different flavors. Since the WMW is not a continuous function various approximations have been used.

A class of these methods have an Support Vector Machine (SVM)-type flavor where the hinge loss is used as a convex upper bound for the 0-1 indicator function [93, 57, 7, 20]. Algorithms similar to the SVMs were used to learn the ranking function.

Another class of methods use a sigmoid [35] or a polynomial approximation [93] to the 0-1 loss function. Similar to our approach they use a gradient based learning algorithm.

8.2.5 Relationship to the current paper

Similar to the papers mentioned, our algorithm is also based on the common approach of trying to correctly arrange pairs of samples, treating them as independent. However our algorithm differs from the previous approaches in the following ways–

- Most of the proposed approaches [33, 40, 8, 34, 16] are computationally expensive to train due to the quadratic scaling in the number of pairwise constraints. While the number of pairwise constraints is quadratic the proposed algorithm is still linear. This is achieved by an efficient algorithm for the fast approximate summation of erfc functions, which allows us to factor the computations.

- There are no approximations in our ranking formulation as in [94], where in order to reduce the quadratic growth a bound on the risk functional is used. It should be noted that we use approximations only in the gradient computation of the optimization procedure. As a result the optimization will converge to the same solution, but will take a few more iterations.

- The other approximate algorithm [23] scales well to large datasets computationally, but it make very coarse approximations by summarizing the slack variables for an entire class by a single, common scalar value.

- The cost function which we optimize is a lower bound on the WMW–the measure which is frequently used to asses the quality of rankings. Previous approaches which try to maximize the WMW [93, 35, 57, 7, 20] consider only

a classification problem and also incur the quadratic growth in the number of constraints.

- Also to optimize our cost function we use the nonlinear conjugate gradient algorithm–which converges much rapidly than the steepest gradient method used for instance by the backpropagation algorithm in RankNet [8].

- In this paper we bring computational tractability to large scale batch optimization algorithms. However the proposed cost function can be optimized using stochastic gradient descent techniques.

8.3 The MAP estimator for learning ranking functions

For ease of exposition we will consider the family of linear ranking functions: $\mathcal{F} = \{f_w\}$, where for any $x, w \in \mathbb{R}^d$, $f_w(x) = w^T x$. A nonlinear version of the algorithm can be easily derived using the *kernel trick* (See [33] for an SVM analog).

Although we want to choose w to maximize the generalized $\text{WMW}(f_w, \mathcal{A}, \mathcal{G})$, for computational efficiency, we shall instead maximize a continuous surrogate, via the log-likelihood:

$$\begin{aligned} \mathcal{L}(f_w, \mathcal{A}, \mathcal{G}) &= \log \Pr\left[\text{correct ranking}|w\right] \\ &= \log \prod_{\mathcal{E}_{ij}} \prod_{k=1}^{m_i} \prod_{l=1}^{m_j} \Pr\left[f_w(x_l^j) > f_w(x_k^i)|w\right]. \end{aligned} \tag{8.4}$$

Note that in Equation 8.4, in common with most papers [8, 33], we have assumed that every pair (x_l^j, x_k^i) is drawn independently, whereas only the original samples

are drawn independently.

We use the sigmoid function to model the pairwise probability, *i.e.*

$$\Pr\left[f_w(x_l^j) > f_w(x_k^i)|w\right] = \sigma\left[w^T(x_l^j - x_k^i)\right], \tag{8.5}$$

$$\text{where} \quad \sigma(z) = \frac{1}{1+e^{-z}} \tag{8.6}$$

is the sigmoid function. The sigmoid function has been previously used in [8] to model pairwise posterior probabilities. However the cost function used was the cross-entropy.

Assuming a prior $\Pr[w] = \mathcal{N}(w|0, \lambda^{-1})$ on the weights w, the optimal *maximum a-posteriori* (MAP) estimator is of the form

$$\widehat{w}_{\text{MAP}} = \arg\max_w L(w), \tag{8.7}$$

where $L(w)$ is the penalized log-likelihood:

$$L(w) = -\frac{\lambda}{2}\|w\|^2 + \sum_{\mathcal{E}_{ij}} \sum_{k=1}^{m_i} \sum_{l=1}^{m_j} \log \sigma\left[w^T(x_l^j - x_k^i)\right]. \tag{8.8}$$

8.3.1 Lower bounding the WMW

Comparing the log-likelihood $L(w)$ to the WMW we can see that this is equivalent to lower bounding the 0-1 indicator function in the WMW by a log-sigmoid function (see Figure 8.2), *i.e.*,

$$\mathbf{1}_{z>0} \geq 1 + (\log \sigma(z)/\log 2). \tag{8.9}$$

The log-sigmoid is appropriately scaled and shifted to make the bound tight at the origin. The log-sigmoid bound was also used in [16] along with a boosting

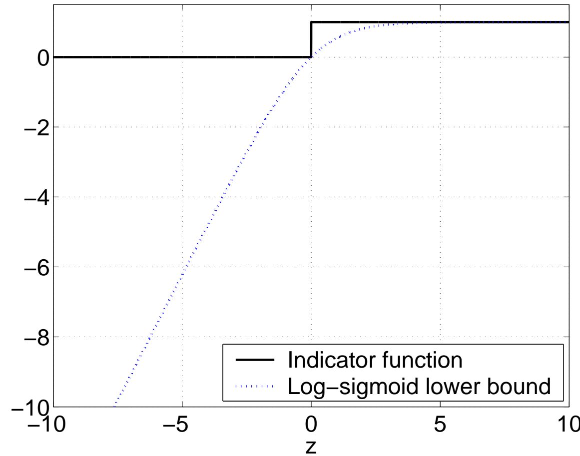


Figure 8.2: Log-sigmoid lower bound for the 0-1 indicator function.

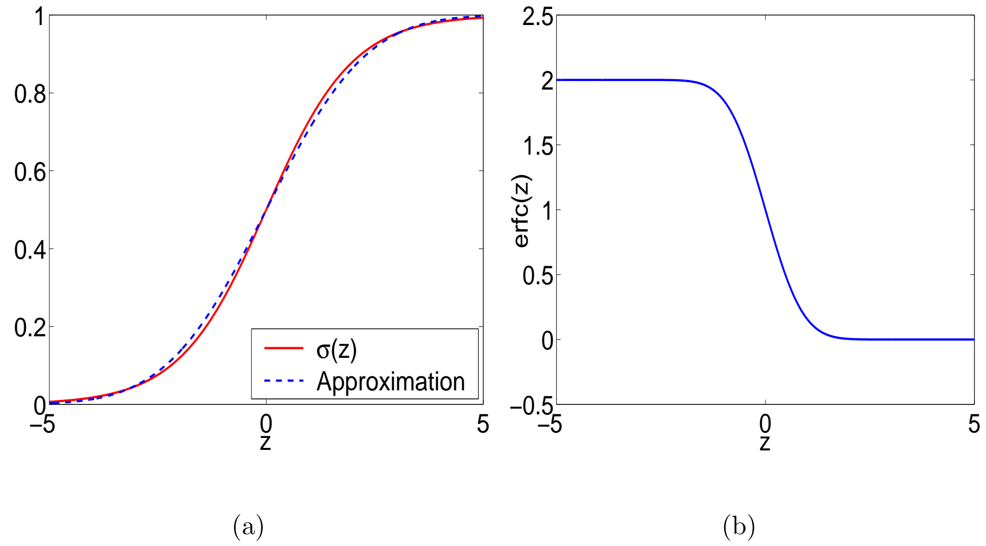
algorithm. So maximizing the penalized log-likelihood is equivalent to maximizing a lower bound on the WMW. The prior $\Pr[w]$ acts as a regularizer.

8.4 The optimization algorithm

In order to find the w that maximizes the penalized log-likelihood, we use the Polak-Ribière variant of nonlinear *conjugate gradients* (CG) algorithm [53]. The CG method only needs the gradient $g(w)$ and does not require evaluation of $L(w)$. It also avoids the need for computing the second derivatives (Hessian matrix). The gradient vector is given by (using the fact that $\sigma^{'}(z) = \sigma(z)\sigma(-z)$ and $\sigma(-z) = 1 - \sigma(z)$):

$$\begin{aligned} g(w) &= \nabla L(w) \\ &= -\lambda w - \sum_{\mathcal{E}_{ij}} \sum_{k=1}^{m_i} \sum_{l=1}^{m_j} (x_k^i - x_l^j) \sigma \left[w^T (x_k^i - x_l^j) \right]. \end{aligned} \tag{8.10}$$

Notice that the evaluation of the penalized log-likelihood or its gradient requires $\mathcal{M}^2 = \sum_{\mathcal{E}_{ij}} m_i m_j$ operations — this quadratic scaling can be prohibitively expensive



(a) (b)

Figure 8.3: (a) Approximation of the sigmoid function $\sigma(z) \approx 1 - \frac{1}{2}\mathrm{erfc}(\frac{\sqrt{3}z}{\sqrt{2}\pi})$. (b) The erfc function.

for large datasets. The main contribution of this paper is an extremely fast method to compute the gradient approximately (Section 8.5).

8.4.1 Gradient approximation using the error-function

We shall rely on the approximation [See Figure 8.3(a)]:

$$\sigma(z) \approx 1 - \frac{1}{2}\mathrm{erfc}(\frac{\sqrt{3}z}{\sqrt{2}\pi}), \tag{8.11}$$

where the complementary error function [Figure 8.3(b)] is defined by

$$\mathrm{erfc}(z) = \frac{2}{\sqrt{\pi}} \int_z^\infty e^{-t^2} dt. \tag{8.12}$$

As a result, the approximate gradient can be computed—still with $\mathcal{O}(d\mathcal{M}^2)$ operations—as:

$$g(w) \approx -\lambda w - \sum_{\mathcal{E}_{ij}} \sum_{k=1}^{m_i} \sum_{l=1}^{m_j} (x_k^i - x_l^j) \left[1 - \frac{1}{2} \text{erfc} \left(\frac{\sqrt{3} w^T (x_k^i - x_l^j)}{\sqrt{2}\pi} \right) \right]. \tag{8.13}$$

8.4.2 Quadratic complexity of gradient evaluation

We will isolate the key computational primitive contributing to the quadratic complexity in the gradient computation. The following summarizes the different variables in analyzing the computational complexity of evaluating the gradient.

- We have S classes with m_i training instances in the ith class.
- Hence we have a total of $m = \sum_{i=1}^{S} m_i$ training examples in d dimensions.
- $|\mathcal{E}|$ is the number of edges in the preference graph, and
- $\mathcal{M}^2 = \sum_{\mathcal{E}_{ij}} m_i m_j$ is the total number of pairwise preference relations.

For any x we will define $z = \sqrt{3} w^T x / (\pi \sqrt{2})$. Note that z is a scalar and for a given w can be computed in $\mathcal{O}(dm)$ operations for the entire training set. We will now rewrite the gradient as

$$g(w) = -\lambda w - \Delta_1 + \frac{1}{2}\Delta_2 - \frac{1}{2}\Delta_3, \tag{8.14}$$

where the vectors Δ_1, Δ_1, and Δ_3 are defined as follows–

$$\begin{aligned}\Delta_1 &= \sum_{\mathcal{E}_{ij}} \sum_{k=1}^{m_i} \sum_{l=1}^{m_j} (x_k^i - x_l^j). \\ \Delta_2 &= \sum_{\mathcal{E}_{ij}} \sum_{k=1}^{m_i} \sum_{l=1}^{m_j} x_k^i \text{erfc}(z_k^i - z_l^j). \\ \Delta_3 &= \sum_{\mathcal{E}_{ij}} \sum_{k=1}^{m_i} \sum_{l=1}^{m_j} x_l^j \text{erfc}(z_k^i - z_l^j). \end{aligned} \tag{8.15}$$

The vector Δ_1 is independent of w and can be written as follows–

$$\Delta_1 = \sum_{\mathcal{E}_{ij}} m_i m_j (x_{\text{mean}}^i - x_{\text{mean}}^j), \text{ where } x_{\text{mean}}^i = \frac{1}{m_i} \sum_{k=1}^{m_i} x_k^i$$

is the mean of all the training instances in the ith class. Hence Δ_1 can be pre-computed in $\mathcal{O}(|\mathcal{E}|d + dm)$ operations.

The the other two terms Δ_2 and Δ_3 can be written as follows–

$$\Delta_2 = \sum_{\mathcal{E}_{ij}} \sum_{k=1}^{m_i} x_k^i E_-^j(z_k^i) \quad \Delta_3 = \sum_{\mathcal{E}_{ij}} \sum_{l=1}^{m_j} x_l^j E_+^i(-z_l^j) \tag{8.16}$$

where

$$\begin{aligned} E_-^j(y) &= \sum_{l=1}^{m_j} \text{erfc}(y - z_l^j). \\ E_+^i(y) &= \sum_{k=1}^{m_i} \text{erfc}(y + z_k^i). \end{aligned} \tag{8.17}$$

Note that $E_-^j(y)$ in the sum of m_j erfc functions centered at z_l^j and evaluated at y–which requires $\mathcal{O}(m_j)$ operations. In order to computed Δ_3 we need to evaluate it at m_i points, thus requiring $\mathcal{O}(m_i m_j)$ operations. Hence each of Δ_2 and Δ_3 can be computed in $\mathcal{O}(dSm + \mathcal{M}^2)$ operations.

Hence the core computational primitive contributing to the $\mathcal{O}(\mathcal{M}^2)$ cost is the summation of erfc functions. In the next section we will show how this sum

can be computed in linear $\mathcal{O}(m_i + m_j)$ time, at the expense of reduced precision which however can be arbitrary. As a result of this Δ_2 and Δ_3 can be computed in linear $\mathcal{O}(dSm + (S-1)m)$ time. In terms of the optimization algorithm since the gradient is computed approximately the number of iterations required to converge will increase. However this is more than compensated by the cost per iteration which is drastically reduced.

8.5 Fast weighted summation of erfc functions

In general $E_-^j(y)$ and $E_+^i(y)$ can be written as the weighted summation of N erfc functions centered at $z_i \in \mathcal{R}$, with weights $q_i \in \mathcal{R}$:

$$E(y) = \sum_{i=1}^{N} q_i \operatorname{erfc}(y - z_i). \tag{8.18}$$

Direct computation of (8.18) at M points $\{y_j \in \mathcal{R}\}_{j=1}^{M}$ is $\mathcal{O}(MN)$. In Chapter 4 we derived an ϵ-exact approximation algorithm to compute this in $\mathcal{O}(M+N)$ time.

For any given $\epsilon > 0$, $\widehat{E}$ is an $\epsilon - exact$ approximation to E if the maximum absolute error relative to the total weight $Q_{abs} = \sum_{i=1}^{N} |q_i|$ is upper bounded by a specified ϵ, i.e.,

$$\max_{y_j} \left[\frac{|\hat{E}(y_j) - E(y_j)|}{Q_{abs}} \right] \leq \epsilon. \tag{8.19}$$

The constant in $\mathcal{O}(M+N)$ for our algorithm depends on the desired accuracy ϵ, which however can be *arbitrary.* In fact, for machine precision accuracy there is no difference between the direct and the fast methods. The algorithm we present is inspired by the fast multipole methods proposed in computational physics [27]. The

fast algorithm is based on using an infinite series expansion for the erfc function and retaining only the first few terms (whose contribution is at the desired accuracy).

8.6 Ranking experiments

8.6.1 Datasets

We used two artificial datasets and ten publicly available benchmark datasets [1] in Table 8.1, previously used for evaluating ranking [23] and ordinal regression [88]. Since these datasets are originally designed for regression, we discretize the continuous target values into S equal sized bins as specified in Table 8.1. For each dataset the number of classes S was chosen such that none of them were empty. The two datasets RandNet and RandPoly are artificial datasets generated as described in [8]. The ranking function for RandNet is generated using a random two layer neural net with 10 hidden units and RandPoly using a random polynomial.

8.6.2 Evaluation procedure

For each data set 80% of the examples were used for training and the remaining 20% were used for testing. The results are shown for a five-fold cross validation experiment. In order to choose the regularization parameter λ, on each fold we used the training split and performed a five-fold cross validation on the training set. The performance is evaluated in terms of the generalized WMW statistic (A WMW of one implies perfect ranking). We used a full order graph to evaluate the ranking

[1] The datasets were downloaded from http://www.liacc.up.pt/~ltorgo/Regression/DataSets.html

performance.

We compare the performance and the time taken for the following methods–

1. *RankNCG* The proposed nonlinear conjugate-gradient ranking procedure. The tolerance for the conjugate gradient procedure was set to 10^{-3} . The nonlinear conjugate gradient optimization procedure was randomly initialized. We compare the following two version–

    - *RankNCG direct* This uses the exact gradient computations.

    - *RankNCG fast* This uses the fast approximate gradient computation. The accuracy parameter ϵ for the fast gradient computation was set to 10^{-6}.

2. *RankNet* [8] A neural network which is trained using pairwise samples based on cross-entropy cost function. For training in addition to the preference relation $x_i \succeq x_j$, each pair also has a associated target posterior $\Pr[x_i \succeq x_j]$. In our experiments we used hard target probabilities of 1 for all pairs. The best learning rate for the net was chosen using WMW as the cross validation measure. Training was done in a batch mode for around 500-1000 epochs or till there are no function decrease in the cost function. We used two version of the RankNet–

    - *RankNet two layer* A two layer neural network with 10 hidden units.

    - *RankNet linear* A single layer neural network.

3. *RankSVM* [40, 33] A ranking function is learnt by training an SVM classifier [2]

[2]Using the SVM-light packages available at http://svmlight.joachims.org/

over pairs of examples. The tradeoff parameter was chosen by cross validation. We used two version of the RankSVM–

- *RankSVM linear* The SVM is trained using a linear kernel.
- *RankSVM quadratic* The SVM is trained using a polynomial kernel $k(x, y) = (x.y + c)^p$ of order $p = 2$.

4. *RankBoost* [20] A boosting algorithm which effectively combines a set of weak ranking functions. We used $\{0, 1\}$-valued weak rankings that use the ordering information provided by the features [20]. Training a weak ranking function involves finding the best feature and the best threshold for that feature. We boosted for 50-100 rounds.

8.6.3 Results

The results are summarized in Table 8.2 and 8.3. All experiments were run on a 1.83GHz machine with 1.00GB of RAM. The following observations can be made.

8.6.3.1 Quality of approximation

The WMW is similar for (a) the proposed exact method (RankNCG direct) (b) the approximate method (RankNCG fast). The run time of the approximate method is one to two magnitudes lower than the exact method, especially for large data sets. Thus we are able to get very good speedups without sacrificing ranking accuracy.

8.6.3.2 Comparison with other methods

All the methods show very similar WMW scores. In terms of the training time the proposed method clearly beats all the other methods. For small datasets RankSVM linear is comparable in time to our methods. For large datasets RankBoost shows the next best time.

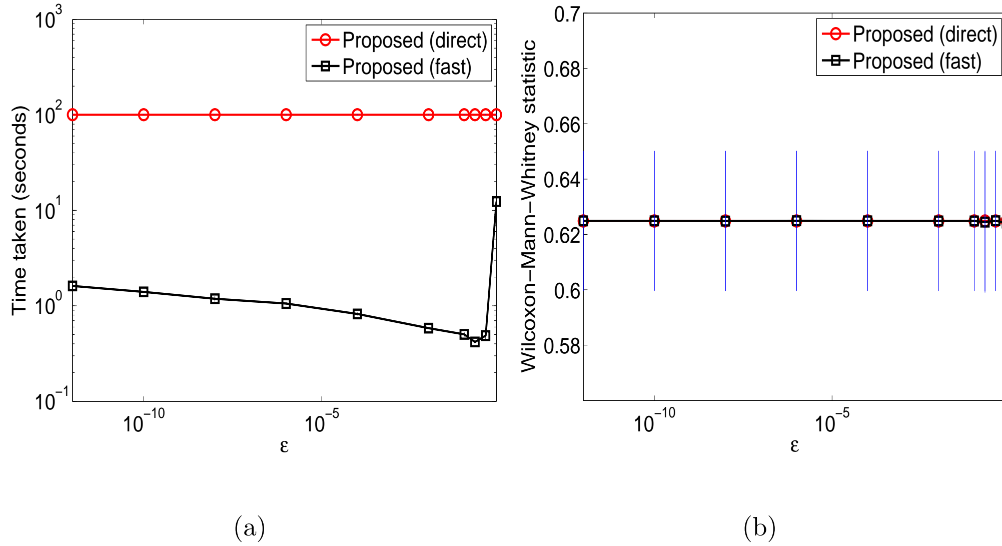
8.6.3.3 Ability to handle large datasets

For dataset 14 only the fast method completed execution. The direct method and all the other methods either crashed due to huge memory requirements or took an incredibly large amount of time. Further, since the accuracy of learning (*i.e.* estimation) clearly depends on the ability to leverage large datasets, in real life, the proposed methods are also expected to be more accurate on large-scale ranking problems.

8.6.4 Impact of the gradient approximation:

Figure 8.4 studies the accuracy and the run-time for dataset 10 as a function of the gradient tolerance, ϵ. As ϵ increases, the time taken per-iteration (and hence overall) decreases. However, if it is too large the total time taken starts increasing (after $\epsilon = 10^{-2}$ in Figure 8.4(a)). Intuitively, this is because the use of approximate derivatives slows the convergence of the conjugate gradient procedure by increasing the number of iterations required for convergence. The speedup is achieved because computing the approximate derivatives is extremely fast, thus compensating for

the slower convergence. However, after a certain point the number of iterations dominates the run-time. Also, notice that ϵ has no significant effect on the WMW achieved, because the optimizer still converges to the optimal value albeit at a slower rate.



(a) (b)

Figure 8.4: *Effect of ϵ-exact derivatives* (a) The time taken and (b) the WMW statistic for the proposed method and the faster version of the proposed method as a function of ϵ. The CG tolerance was set to 10^{-3}. Results are for dataset 10.

Table 8.1: Benchmark datasets used in the ranking experiments. N is the size of the data set. d is the number of attributes. S is the number of classes. $\mathcal{M}$ is the average total number of pairwise relations per fold of the training set.

	Dataset name	N	d	S	$\mathcal{M}$		Dataset name	N	d	S	$\mathcal{M}$
1	Diabetes	43	3	2	272	8	Airplane Companies	950	10	5	217301
2	Pyrimidines	74	28	3	1113	9	RandNet	1000	50	6	195907
3	Triazines	186	61	4	7674	10	RandPoly	1000	50	6	225131
4	Wisconsin Breast Cancer	194	33	4	8162	11	Abalone	4177	9	3	3713729
5	Machine-CPU	209	7	4	9820	12	RandNet	5000	50	6	6269910
6	Auto-MPG	392	8	3	30057	13	RandPoly	5000	50	6	5367241
7	Boston Housing	506	14	2	33693	14	California Housing	20640	9	3	82420255

Table 8.2: The mean training time and standard deviation in seconds for the various methods and all the datasets shown in Table 8.1. The results are shown for a five fold cross-validation experiment. The symbol $\star$ indicates that the particular method either crashed due to limited memory requirements or took a very large amount of time.

	RankNCG direct	**RankNCG fast**	RankNet linear	RankNet two layer	RankSVM linear	RankSVM quadratic	RankBoost
1	0.11 [± 0.02]	0.06 [± 0.01]	1.79 [± 0.03]	3.32 [± 0.11]	0.09 [± 0.04]	0.10 [± 0.01]	1.70 [± 0.09]
2	0.63 [± 0.13]	0.12 [± 0.03]	7.11 [± 0.27]	13.55 [± 0.30]	0.10 [± 0.02]	0.62 [± 0.13]	1.72 [± 0.02]
3	17.63 [± 7.27]	0.70 [± 0.39]	58.14 [± 0.78]	131.41 [± 2.19]	0.55 [± 0.28]	13.96 [± 0.48]	6.70 [± 0.06]
4	13.41 [± 9.35]	0.33 [± 0.43]	48.13 [± 0.85]	97.24 [± 1.05]	0.64 [± 0.03]	23.17 [± 3.37]	1.88 [± 0.04]
5	20.38 [± 4.87]	0.97 [± 0.15]	57.99 [± 0.58]	111.14 [± 1.14]	1.14 [± 0.27]	24.46 [± 0.68]	1.24 [± 0.02]
6	28.05 [± 10.94]	0.40 [± 0.23]	175.63 [± 1.55]	333.49 [± 3.96]	0.43 [± 0.02]	37.27 [± 3.10]	1.54 [± 0.04]
7	18.92 [± 0.63]	0.16 [± 0.01]	195.14 [± 4.75]	381.28 [± 7.93]	0.36 [± 0.03]	13.93 [± 2.15]	2.32 [± 0.04]
8	332.88 [± 26.66]	3.29 [± 0.88]	1264.58 [± 3.21]	2464.84 [± 10.94]	34.32 [± 4.05]	1332.79 [± 69.47]	5.56 [± 0.37]
9	250.37 [± 21.03]	5.08 [± 0.47]	1166.23 [± 17.47]	2380.62 [± 34.53]	83.62 [± 6.30]	13628.23 [± 210.10]	13.55 [± 0.07]
10	102.48 [± 0.59]	0.78 [± 0.04]	1341.20 [± 6.91]	2733.25 [± 23.11]	1656.52 [± 99.89]	14110.48 [± 121.98]	13.99 [± 0.05]
11	1736.47 [± 191.03]	1.47 [± 0.38]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	62.91 [± 0.59]
12	6731.09 [± 312.41]	19.10 [± 1.76]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	147.04 [± 0.16]
13	2556.93 [± 15.03]	3.59 [± 0.41]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	133.42 [± 1.14]
14	⋆ [± ⋆]	46.86 [± 1.06]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]

Table 8.3: The corresponding generalized Wilcoxon-Mann-Whitney statistic on the test set for the results shown in Table 8.2.

	RankNCG direct	**RankNCG fast**	RankNet linear	RankNet two layer	RankSVM linear	RankSVM quadratic	RankBoost
1	0.677 [± 0.233]	0.650 [± 0.210]	0.579 [± 0.096]	0.479 [± 0.284]	0.545 [± 0.236]	0.400 [± 0.276]	0.675 [± 0.173]
2	0.987 [± 0.019]	0.948 [± 0.077]	0.872 [± 0.088]	0.968 [± 0.038]	0.973 [± 0.048]	0.837 [± 0.142]	0.906 [± 0.144]
3	0.942 [± 0.044]	0.914 [± 0.047]	0.828 [± 0.030]	0.891 [± 0.064]	0.934 [± 0.019]	0.861 [± 0.088]	0.651 [± 0.045]
4	0.764 [± 0.028]	0.771 [± 0.046]	0.773 [± 0.046]	0.750 [± 0.035]	0.793 [± 0.018]	0.795 [± 0.035]	0.748 [± 0.056]
5	0.920 [± 0.015]	0.938 [± 0.020]	0.919 [± 0.035]	0.923 [± 0.040]	0.929 [± 0.026]	0.901 [± 0.014]	0.926 [± 0.018]
6	0.999 [± 0.002]	0.998 [± 0.002]	0.998 [± 0.003]	0.996 [± 0.003]	0.998 [± 0.002]	0.995 [± 0.008]	0.992 [± 0.004]
7	1.000 [± 0.000]	1.000 [± 0.000]	1.000 [± 0.000]	0.800 [± 0.400]	1.000 [± 0.000]	1.000 [± 0.000]	1.000 [± 0.000]
8	0.984 [± 0.004]	0.984 [± 0.003]	0.951 [± 0.004]	0.765 [± 0.245]	0.984 [± 0.004]	0.996 [± 0.001]	0.958 [± 0.003]
9	0.944 [± 0.012]	0.944 [± 0.012]	0.915 [± 0.017]	0.899 [± 0.028]	0.945 [± 0.013]	0.747 [± 0.005]	0.848 [± 0.015]
10	0.625 [± 0.025]	0.625 [± 0.025]	0.688 [± 0.032]	0.644 [± 0.054]	0.625 [± 0.026]	0.823 [± 0.008]	0.618 [± 0.024]
11	0.536 [± 0.011]	0.534 [± 0.008]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	0.535 [± 0.014]
12	0.917 [± 0.005]	0.917 [± 0.005]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	0.845 [± 0.006]
13	0.623 [± 0.008]	0.623 [± 0.008]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	0.607 [± 0.010]
14	⋆ [± ⋆]	0.979 [± 0.001]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]	⋆ [± ⋆]

8.7 Application to Collaborative filtering

As an application we will show some results on a collaborative filtering task for movie recommendations. We use the MovieLens dataset [3] which contains approximately 1 million ratings for 3592 movies by 6040 users. Ratings are made on a scale of 1 to 5. The task is to predict the movie rankings for a user based on the rankings provided by other users. For each user we used 70% of the movies rated by him for training and the remaining 30% for testing. The features for each movie consisted of the ranking provided by d other users. For each missing rating, we imputed a sample drawn from a Gaussian distribution with its mean and variance estimated from the available ratings provided by the other users. Table 8.4 and 8.5 shows the time taken and the WMW score for this task for the two fastest methods. The results are averaged over 100 users. The other methods took a large amount of time to train just for one user. The proposed method shows the best WMW and takes the least amount of time for training.

8.8 Conclusion and future work

We presented an approximate ranking algorithm which directly maximizes the generalized Wilcoxon-Mann-Whitney statistic. The algorithm was made computationally tractable using a novel, fast summation method for calculating a weighted sum of erfc functions. Experimental results demonstrate that despite the order of magnitude speedup, the accuracy was almost identical to exact method and other

[3] The dataset was downloaded from http://www.grouplens.org/.

Table 8.4: Results for the EACHMOVIE dataset: The mean training time (averaged over 100 users) as a function of the number of features d.

d	RankNCG fast	RankBoost
50	0.48 [± 0.19]	6.68 [± 1.65]
100	0.44 [± 0.17]	12.67 [± 2.83]
200	0.42 [± 0.17]	27.53 [± 5.99]
400	0.41 [± 0.17]	68.08 [± 13.95]
800	0.45 [± 0.13]	193.18 [± 39.75]
1600	0.51 [± 0.15]	613.54 [± 124.93]

Table 8.5: The corresponding generalized WMW statistic on the test set for the results shown in Table 8.4.

d	RankNCG fast	RankBoost
50	0.693 [± 0.054]	0.672 [± 0.056]
100	0.707 [± 0.049]	0.679 [± 0.050]
200	0.722[± 0.053]	0.685 [± 0.057]
400	0.720 [±0.054]	0.685 [± 0.051]
800	0.721 [± 0.050]	0.673 [± 0.058]
1600	0.719 [± 0.053]	0.682 [± 0.058]

algorithms proposed in literature.

8.8.1 Future Work

Other applications for fast summation of erfc functions: The fast summation method proposed could be potentially useful in neural networks, probit regression, and in Bayesian models involving sigmoids.

Nonlinear, kernelized variations: In order to retain focus, we did not discuss the non-linear version of our algorithm in detail. However, we may easily kernelize it by replacing the linear function $w^T x$ with $\sum_{i=1}^{m} k(x, x_i)\alpha_i = \alpha^T \mathbf{k}(x)$, where k is the kernel used and $\mathbf{k}(x)$ is a column vector defined by $\mathbf{k}(x) = [k(x, x_1), \ldots, k(x, x_m)]^T$. The penalized log-likelihood for this problem changes to:

$$L(\alpha) = -\frac{\lambda}{2}\|\alpha\|^2 + \sum_{\mathcal{E}_{ij}} \sum_{k=1}^{m_i} \sum_{l=1}^{m_j} \log \sigma \left[\alpha^T \left(\mathbf{k}(x_l^j) - \mathbf{k}(x_k^i)\right)\right]. \qquad (8.20)$$

The gradient vector is given by:

$$\begin{aligned} g(\alpha) = \nabla L(\alpha) &= -\lambda\alpha \\ &- \sum_{\mathcal{E}_{ij}} \sum_{k=1}^{m_i} \sum_{l=1}^{m_j} \left(\mathbf{k}(x_k^i) - \mathbf{k}(x_l^j)\right) \sigma \left[\alpha^T \left(\mathbf{k}(x_k^i) - \mathbf{k}(x_l^j)\right)\right]. \end{aligned} \qquad (8.21)$$

The computation of the gradient will involve calculating: (a) the weighted sum of kernel functions, and (b) the weighted sum of sigmoid (or erfc) functions. Dual-tree methods [26] and the improved fast Gauss transform [95] may be used to speedup (a). For (b) we can use the fast approximation proposed in this paper.

Independence of pairs of samples: In common with most papers following [33], we have assumed that every pair (x_l^j, x_k^i) is drawn independently, even though they

are really correlated (actually, the samples x_k^i are drawn independently). In the future we plan to correct for this lack of independence using a statistical random-effects-model.

Effect of ϵ on convergence rate: We plan to study the convergence behavior of the conjugate gradient procedure using approximate gradient computations. This would give us a formal mechanism to choose ϵ.

Chapter 9

Conclusions

This thesis introduced a new mathematical tool for fast summation–a fundamental type of computational problem occurring widely in machine learning. Specifically we designed three fast summation algorithms and applied it to a few machine learning tasks. The source code for all the fast summation algorithms are released under the Lesser GPL.

9.1 Future work

The following problems are among those that I wish to formulate well and solve in the future.

- ***Core algorithms*** Development of these kind of fast approximate algorithms for more kernels–e.g., the Epanechnikov kernel for kernel density estimation and the Matèrn class of kernels used in Gaussian process regression.

- ***Convergence issues*** In many applications these fast MVP primitives are embedded in a optimization routine–e.g., in ranking problem we embedded it in a conjugate-gradient procedure. A theoretical issue which we have barely touched upon concerns the convergence of these optimization routines when using approximate MVP primitives.

- ***Applications*** A few applications which I would like to further explore include– hyperparameter selection for Gaussian processes, Nadarya-Watson kernel regression, and inexact eigenvalue methods for unsupervised learning.

9.2 Open problems

Following are a few open problems which may require much time and thought.

- ***The curse of dimensionality.*** For the Gaussian kernel our experimental results indicate that it easy to get good speedups at very large or very small bandwidths. For moderate bandwidths and moderate dimensions ($d \leq 10$) our proposed algorithm is capable of giving good speedups. However getting good speedups for *moderate bandwidths and large dimensions* remains an important open research problem.

- ***The paradox of the curse of dimensionality.*** For most machine learning tasks even though the data is very high dimensional, the true intrinsic dimensionality is typically very small. I intend to explore if dimensionality reduction approaches like PCA and manifold learning methods can be directly incorporated into our fast algorithms.

- ***Structure, Inference, and Computation*** A more ambitious task would be to explore if there are any deeper connections between structure in the data, computation, and inference.

Bibliography

[1] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* Dover, 1972.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching. *Journal of the ACM*, 45:891–923, 1998.

[3] B. J. C. Baxter and G. Roussos. A new error estimate of the fast gauss transform. *SIAM Journal of Scientific and Statistical Computing*, 24(1):257–259, 2002.

[4] N. C. Beauliu. A simple series for personal computer computation of the error function $Q(.)$. *IEEE Transactions on Communications*, 37(9):989–991, 1989.

[5] M. Bern and D. Eppstein. *Approximation algorithms for NP-hard problems*, chapter Approximation algorithms for geometric problems, pages 296–345. PWS Publishing Company, Boston, 1997.

[6] P. K. Bhattacharya. Estimation of a probability density function and its derivatives. *Sankhya, Series A*, 29:373–382, 1967.

[7] U. Brefeld and T. Scheffer. AUC maximizing support vector learning. In *Proceedings of the ICML 2005 Workshop on ROC Analysis in Machine Learning*, 2005.

[8] C.J.C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. Learning to rank using gradient descent. In *Proceeding of the* 22nd *International conference on Machine Learning*, 2005.

[9] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. Reconstruction and representation of 3d objects with radial basis functions. In *ACM SIGGRAPH 2001*, pages 67–76, Los Angeles, CA, 2001.

[10] R. Caruana, S. Baluja, and T. Mitchell. Using the future to 'sort out' the present: Rankprop and multitask learning for medical risk evaluation. In *Advances in Neural Information Processing Systems*, 1995.

[11] F.R.K. Chung. *Spectral Graph Theory.* Amer. Math. Society Press, 1997.

[12] K. Crammer and Y. Singer. Pranking with ranking. *Advances in Neural Information Processing Systems*, 14:641–647, 2002.

[13] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines (and other kernel-based learning methods).* Cambridge University Press, 2000.

[14] L. Csato and M. Opper. Sparse on-line gaussian processes. *Neural Computation*, 14(3):641–668, 2002.

[15] N. De Freitas, Y. Wang, M. Mahdaviani, and D. Lang. Fast Krylov methods for N-body learning. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems 18*. MIT Press, Cambridge, MA, 2006.

[16] O. Dekel, C. Manning, and Y. Singer. Log-linear models for label ranking. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA, 2004.

[17] J. Dongarra and F. Sullivan. The top ten algorithms of the century. *Computing in Science and Engineering*, 2(1):22–23, 2000.

[18] T. Feder and D. Greene. Optimal algorithms for approximate clustering. In *Proc. 20th ACM Symp. Theory of Computing*, pages 434–444, 1988.

[19] S. Fine and K. Scheinberg. Efficient SVM training using low-rank kernel representations. *Journal of Machine Learning Research*, 2:243264, December 2001.

[20] Y. Freund, R. Iyer, and R. Schapire. An efficient boosting algorithm for combining preferences. *Journal of Machine Learning Research*, 4:933–969, 2003.

[21] H. Friedman, J and J. W. Tukey. A projection pursuit algorithm for exploratory data analysis. *IEEE Transactions on Computing*, 23:881–889, 1974.

[22] K. Fukunaga and L. Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on Information Theory*, 21(1):32–40, 1975.

[23] G. Fung, R. Rosales, and B. Krishnapuram. Learning rankings via convex hull separation. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems 18*. MIT Press, Cambridge, MA, 2006.

[24] T. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[25] A. Gray and A. Moore. N-body problems in statistical learning. In *Advances in Neural Information Processing Systems*, pages 521–527, 2001.

[26] A. G. Gray and A. W. Moore. Nonparametric density estimation: Toward computational tractability. In *SIAM International conference on Data Mining*, 2003.

[27] L. Greengard. Fast algorithms for classical physics. *Science*, 265(5174):909–914, 1994.

[28] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. of Comp. Physics*, 73(2):325–348, 1987.

[29] L. Greengard and J. Strain. The fast Gauss transform. *SIAM Journal of Scientific and Statistical Computing*, 12(1):79–94, 1991.

[30] L. Greengard and X. Sun. A new version of the fast gauss transform. *Documenta Mathematica*, Extra Volume ICM(III):575 – 584, 1998.

[31] L. F. Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems.* The MIT Press, 1988.

[32] E. F. Harrington. Online ranking/collaborative filtering using the perceptron algorithm. In *roceedings of the Twentieth International Conference on Machine Learning*, 2003.

[33] R. Herbrich, T. Graepel, P. Bollmann-Sdorra, and K. Obermayer. Learning preference relations for information retrieval. *ICML-98 Workshop: Text Categorization and Machine Learning*, pages 80–84, 1998.

[34] R. Herbrich, T. Graepel, and K. Obermayer. *Advances in Large Margin Classifiers*, chapter Large margin rank boundaries for ordinal regression, pages 115–132. MIT Press, 2000.

[35] A. Herschtal and B. Raskutti. Optimising area under the ROC curve using gradient descent. In *Proceeding of the 21st International conference on Machine Learning*, 2004.

[36] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, 1952.

[37] D. S. Hochbaum and D. B. Shmoys. A best possible heuristic for the k-center problem. *Mathematics of Operations Research*, 10:180–184, 1985.

[38] P. J. Huber. Projection pursuit. *The Annals of Statistics*, 13:435–475, 1985.

[39] A. J. Izenman. Recent developments in nonparametric density estimation. *Journal of American Staistical Association*, 86(413):205–224, 1991.

[40] T. Joachims. Optimizing search engines using clickthrough data. *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 133–142, 2002.

[41] M. C. Jones, J. S. Marron, and S. J. Sheather. A brief survey of bandwidth selection for density estimation. *Journal of American Statistical Association*, 91(433):401–407, March 1996.

[42] M. C. Jones and R. Sibson. What is projection pursuit? *Journal of Royal Statististical Society Series A*, 150:1–36, 1987.

[43] C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations.* SIAM, 1995.

[44] N. Lawrence, M. Seeger, and R. Herbrich. *Advances in Neural Information Processing Systems 15*, chapter Fast Sparse Gaussian Process methods: The Informative Vector Machine, pages 625–632. MIT Press, 2003.

[45] Y.-J. Lee and O. Mangasarian. Rsvm: Reduced support vector machines. In *First SIAM International Conference on Data Mining, Chicago*, 2001.

[46] D. MacKay and M. N. Gibbs. Efficient implementation of Gaussian processes. unpublished, 1997.

[47] D. J. C. MacKay. A practical Bayesian framework for backpropagation networks. *Neural Computation*, 4:448–472, 1992.

[48] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[49] J. S. Marron and M. P. Wand. Exact mean integrated squared error. *The Annals of Statistics*, 20(2):712–736, 1992.

[50] A. Mas-Colell, M.D. Whinston, and J.R. Green. *Microeconomic theory*. Oxford University Press, New York, 1995.

[51] D. M. Mount and S. Arya. ANN: A library for approximate nearest neighbor searching. http://www.cs.umd.edu/ mount/ANN/.

[52] D. J. Newman, S. Hettich, C. L. Blake, and C. J. Merz. UCI repository of machine learning databases. http://www.ics.uci.edu/∼mlearn/MLRepository.html, 1998.

[53] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer-Verlag, 1999.

[54] E. Parzen. On estimation of a probability density function and mode. *Annals of Mathematical Statistics*, 33(3):1065–1076, 1962.

[55] T. Poggio and S. Smale. The mathematics of learning: Dealing with data. *Notices of the American Mathematical Society*, 50(5):537–544, 2003.

[56] J. Quiñonero Candela and C. E. Rasmussen. A unifying view of sparse approximate gaussian process regression. *Journal of Machine Learning Research*, 6:1935–1959, 2005.

[57] A. Rakotomamonjy. Optimizing area under the ROC curve with SVMs. In *ROC Analysis in Artificial Intelligence*, pages 71–80, 2004.

[58] C. E. Rasmussen and C. K. I. Williams. *Gaussian processes for machine learning*, chapter Approximation methods for large datasets, pages 171–188. The MIT Press, 2006.

[59] V. C. Raykar and R. Duraiswami. The improved fast Gauss transform with applications to machine learning. 2005. Presented at the NIPS 2005 workshop on Large scale kernel machines.

[60] V. C. Raykar and R. Duraiswami. Very fast optimal bandwidth selection for univariate kernel density estimation. Technical Report CS-TR-4774, Department of computer science, University of Maryland, Collegepark, 2005.

[61] V. C. Raykar and R. Duraiswami. Fast optimal bandwidth selection for kernel density estimation. In J. Ghosh, D. Lambert, D. Skillicorn, and J. Srivastava, editors, *Proceedings of the sixth SIAM International Conference on Data Mining*, pages 524–528, 2006.

[62] V. C. Raykar and R. Duraiswami. Fast large scale Gaussian process regression using approximate matrix-vector products. *Presented at the Learning workshop 2007, San Juan, Peurto Rico.*, 2007.

[63] V. C. Raykar and R. Duraiswami. Fast weighted summation of erfc functions. Technical Report CS-TR-4848, Department of computer science, University of Maryland, Collegepark, 2007.

[64] V. C. Raykar and R. Duraiswami. *Large Scale Kernel Machines*, chapter The Improved Fast Gauss Transform with applications to machine learning. MIT Press, 2007.

[65] V. C. Raykar, R. Duraiswami, and N. Gumerov. Fast computation of sums of Gaussians. *Journal of Machine Learning Research (submitted)*, 2007.

[66] V. C. Raykar, R. Duraiswami, and B. Krishnapuram. A fast algorithm for learning large scale preference relations. In M. Meila and X. Shen, editors, *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics*, pages 385–392, 2007.

[67] V. C. Raykar, R. Duraiswami, and B. Krishnapuram. A fast algorithm for learning the large scale ranking function. *IEEE Transactions on Pattern Analysis and Machine Intelligence (submitted)*, 2007.

[68] V. C. Raykar, C. Yang, R. Duraiswami, and N. Gumerov. Fast computation of sums of Gaussians in high dimensions. Technical Report CS-TR-4767, Department of Computer Science, University of Maryland, CollegePark, 2005.

[69] B. Schölkopf, J. Giesen, and S. Spalinger. Kernel methods for implicit surface modeling. In Lawrence K. Saul, Yair Weiss, and Léon Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 1193–1200. MIT Press, Cambridge, MA, 2005.

[70] E. F. Schuster. Estimation of a probability density function and its derivatives. *The Annals of Mathematical Statistics*, 40(4):1187–1195, August 1969.

[71] J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

[72] S.J. Sheather and M.C. Jones. A reliable data-based bandwidth selection method for kernel density estimation. *Journal of Royal Statistical Society Series B*, 53(3):683–690, 1991.

[73] Y. Shen, A. Ng, and M. Seeger. Fast Gaussian process regression using KD-trees. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems 18*. MIT Press, Cambridge, MA, 2006.

[74] B. W. Silverman. Algorithm AS 176: Kernel density estimation using the fast Fourier transform. *Journal of Royal Statistical society Series C: Applied statistics*, 31(1):93–99, 1982.

[75] V. Simoncini and D. B. Szyld. Theory of inexact Krylov subspace methods and applications to scientific computing. *SIAM J. Sci. Comput.*, 25(2):454–477, 2004.

[76] R. S. Singh. Applications of estimators of a density and its derivatives to certain statistical problems. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(3):357–363, 1977.

[77] A. Smola and B. Bartlett. Sparse greedy gaussian process regression. In *Advances in Neural Information Processing Systems*, page 619625. MIT Press, 2001.

[78] A. Smola, B. Scholkopf, and K.-R. Muller. Nonlinear component analysis as a kernel eigenvalue problem. Technical Report 44, Max-Planck-Institut fr biologische Kybernetik, Tubingen, 1996.

[79] E. Snelson and Z. Ghahramani. Sparse Gaussian processes using pseudo-inputs. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems 18*. MIT Press, Cambridge, MA, 2006.

[80] C. Tellambura and A. Annamalai. Efficient computation of erfc(x) for large arguments. *IEEE Transactions on Communications*, 48(4):529–532, 2000.

[81] M. Tipping. Sparse bayesian learning and the relevance vector machine. *Journal of machine learning research*, 1:211–244, 2001.

[82] V. Tresp. A bayesian committee machine. *Neural Computation*, 12(11):2719–2741, 2000.

[83] P. M. Vaidya. An optimal algorithm for the all-nearest-neighbors problem. In *Proc. 27th IEEE FOCS*, pages 117–122, 1986.

[84] G. Wahba. *Spline Models for Observational data*. SIAM, 1990.

[85] C. Walder, O. Chapelle, and B. Schölkopf. Implicit surface modelling as an eigenvalue problem. In *Proceedings of the 22nd International Conference on Machine Learning*, pages 937 – 944, 2005.

[86] M. P. Wand and M. C. Jones. Multivariate plug-in bandwidth selection. *Computational Statistics*, 9:97–117, 1994.

[87] M. P. Wand and M. C. Jones. *Kernel Smoothing*. Chapman and Hall, London, 1995.

[88] C. Wei and Z. Ghahramani. Gaussian Processes for Ordinal Regression. *The Journal of Machine Learning Research*, 6:1019–1041, 2005.

[89] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, December 1945.

[90] C. K. I. Williams and C. E. Rasmussen. Gaussian processes for regression. In *Advances in Neural Information Processing Systems*, volume 8, 1996.

[91] C. K. I. Williams and M. Seeger. Using the Nyström method to speed up kernel machines. In *Advances in Neural Information Processing Systems*, page 682688. MIT Press, 2001.

[92] C. K. I. Willimas and D. Barber. Bayesian classification with Gaussian processes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(12):1342–1351, 1998.

[93] L. Yan, R. Dodier, M. Mozer, and R. Wolniewicz. Optimizing classifier performance via an approximation to the Wilcoxon-Mann-Whitney statistic. In *Proceeding of the 20th International conference on Machine Learning*, pages 848–855, 2003.

[94] R. Yan and A. Hauptmann. Efficient margin-based rank learning algorithms for information retrieval. In *International Conference on Image and Video Retrieval (CIVR'06)*, 2006.

[95] C. Yang, R. Duraiswami, and L. Davis. Efficient kernel machines using the improved fast Gauss transform. In L. K. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 1561–1568. MIT Press, Cambridge, MA, 2005.

[96] C. Yang, R. Duraiswami, N. Gumerov, and L. Davis. Improved fast Gauss transform and efficient kernel density estimation. In *IEEE International Conference on Computer Vision*, pages 464–471, 2003.