

Efficient Approximation Methods for Harmonic Semi-Supervised Learning

Andreas Argyriou

MSc Intelligent Systems

University College London

Project Supervisor: Dr. Zoubin Ghahramani¹

September 10, 2004

¹This report is submitted as part requirement for the MSc Degree in Intelligent Systems at University College London. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Contents

1	Introduction	1
2	Semi-Supervised Learning and Approximation Methods	3
2.1	Gaussian Fields and Harmonic Functions	3
2.2	Kd-trees	5
2.3	Conjugate Gradients	7
3	Weight Matrix Construction	11
3.1	Weighting Schemes	11
3.2	Kd-trees in High Dimensions	13
3.3	ϵ NN Search	19
3.4	Effect of the Pivoting Method	20
3.5	Implementation	22
4	Solving the Harmonic Equation	26
4.1	Sparse Linear System Solution	26
4.2	Iterative Methods	27
4.3	Preconditioning	34
4.4	Weight Learning	37
5	Experimental Results	39
5.1	Weight Matrix Construction	39

5.2	Classification	41
5.3	Weights Learning	41
6	Conclusion	46
	References	46

List of Figures

3.1	Sample from the ‘1-2’ data set.	14
3.2	Histogram of variances of values at every pixel.	14
3.3	Range of values along the 12 principal components.	16
3.4	Histogram of distances in the subspace of the 12 principal components.	16
3.5	Computational cost of kd-NN versus sample size with a data set projected on 12 (left) and 20 (right) principal components. Maximum variance was used for pivot selection.	17
3.6	Histogram of the 10 nearest neighbour distances for all points. Nearest neighbours were selected from the original space (left) and from the space of the 12 principal components (right). . .	18
3.7	Logarithm of the number of ϵ nearest neighbours of images of ‘1’ and of images of ‘2’, versus ϵ	19
3.8	Computational cost of kd-NN versus sample size, on 12 principal components using two different pivoting strategies. . . .	21
4.1	Accuracy $\frac{\ \tilde{x}-x\ }{\ x\ }$ (top) and relative residual (bottom) for various iterative methods using Δ_{uu} as coefficient matrix. The stopping criterion is a 10^{-6} tolerance for the relative residual. In all cases, the algorithms converged within this tolerance. . .	30
4.2	Accuracy (top) and relative residual (bottom) for various iterative methods using $I - P_{uu}$ as coefficient matrix.	31

4.3	Required time for various iterative methods using Δ_{uu} (top) and $I - P_{uu}$ (bottom) as coefficient matrix.	32
4.4	Condition number of Δ_{uu} (top) and $I - P_{uu}$ (bottom).	33
4.5	Required time for iterative methods after preconditioning. The systems are the same as in Figures 4.1-4.3 and CGS is also shown for comparison. Jacobi and incomplete LU preconditioning were combined with CGS on $I - P_{uu}$, while incomplete Cholesky preconditioning with MINRES was applied to Δ_{uu}	36
5.1	Cost of constructing a weight matrix from the ‘0-9’ data set (4000 points). On top, the matrix was exp-weighted kNN and at the bottom exp-weighted ϵ NN. Squares correspond to a standard sequential algorithm and dots to the kd-tree algorithm.	40
5.2	Classification error versus time cost for preconditioned CGS and the standard linear system solver. On top, the ‘1-2’ data set is used and at the bottom, the ‘0-9’ data set (classifying digit ‘5’). The graphs were exp-weighted kNN with $k = 10$ and the tolerance of CGS was 10^{-6}	42
5.3	Classification error versus time cost for preconditioned CGS with 20000 points from the MNIST data set (classifying digit ‘3’). On top, unweighted kNN and at the bottom, exp-weighted kNN ($k = 3$). l ranges from 100 to 900. Length scales were equal to 400 and the tolerance of CGS was 10^{-6}	43
5.4	Accuracy versus l for preconditioned CGS with 20000 points from the MNIST data set (classifying digit ‘3’). On top, unweighted kNN and at the bottom, exp-weighted kNN ($k = 3$). Length scales were equal to 400 and the tolerance of CGS was 10^{-6}	44

5.5 Entropy versus time for two weight learning runs, one with
CGS and one with the standard direct solver. 45

List of Tables

2.1	Kd-tree construction algorithm	6
2.2	Kd-tree nearest neighbour search	8
3.1	Weight matrix construction algorithm	23
3.2	k nearest neighbours algorithm	24
3.3	ϵ nearest neighbours algorithm	25

Abstract

An algorithm for semi-supervised learning using Gaussian fields and harmonic functions has been recently proposed by Zhu, Ghahramani and Lafferty. In this approach, a Gaussian field is defined on a weighted graph representing degrees of similarity between data points. A limited number of the points are labeled and it is shown that the unlabeled points can be classified according to the solution of a linear system involving the Laplacian of the graph. Thus, learning reduces to solving several systems of the form $Ax = b$, where A is a fixed symmetric, positive definite matrix.

The objective of this work is to construct the weight matrix of the graph in an efficient way and apply numerical methods to the solution of the linear systems. Principal Component Analysis is used to compress the weight information to a smaller dimensionality. Then, kd-tree nearest neighbour algorithms are applied to form the k or ϵ -neighbourhoods around each point in $O(n \log n)$ time. The result is a sparse weight matrix and its corresponding Laplacian matrix. Classification and hyperparameter learning are then accelerated by using iterative methods for linear systems. These methods, in combination with a simple preconditioning technique, cost much less than the superquadratic direct methods and enable the application of the learning algorithm to larger data sets. Finally, the improvements in performance are demonstrated by some real-life experiments.

Acknowledgments

I would like to thank Dr. Zoubin Ghahramani for supervising this project; Xiaojin Zhu for suggesting the topic; Dr. Massimiliano Pontil for useful discussions; and P. Boufounos, C. Antoniou and P. Smaragdis for technical advice and hints.

Chapter 1

Introduction

Machine learning algorithms can be categorised in two main types. *Supervised learning* algorithms are aided by given labeled examples while *unsupervised learning* algorithms exploit the structure of the data in the absence of labeled examples. Recently, however, there has been interest in cases in which there are labeled examples but they are relatively few. These problems are called *semi-supervised* and they have given rise to several new algorithms. For example, there have been approaches that compute a manifold embedding and graph-based approaches. In (Zhu et al., 2003a), an approach inspired by graphical models is presented and in (Zhu et al., 2003b) it is generalised inside the framework of Gaussian Processes. This approach, assumes a Gaussian distribution over the labels of data points and, like other semi-supervised algorithms, involves the combinatorial Laplacian of the graph. The resulting expression is a concise linear system whose solution gives the labels for the unlabeled data points based on the known labels. The result can be viewed as a kind of harmonic function, meaning that it averages neighbouring values.

This solution depends on the choice of the graph weights. Intuitively, these should reflect the proximity of two points, but it is not necessary that a Euclidean distance is the right choice. There are several possibilities for

weighting schemes, some of them depending on a number of hyperparameters. To learn the appropriate weighting and its hyperparameters, a maximum likelihood procedure could be used. However, in (Zhu et al., 2003a) a simpler entropy minimisation approach is followed and it leads to a gradient descent in the hyperparameter space.

Still, the computational cost of learning the hyperparameters is prohibitive with data sets more than a few thousand points, if an exact solution is sought and the full graph information is used. This motivates us to find ways of approximating the solution in less time. To achieve this, it can be assumed that a few nearest neighbours around each point approximate well enough the contribution of the graph weights to the harmonic function. Then, fast iterative methods can be applied to the resulting sparse linear systems. In the following, I present an implementation of such an approximation for harmonic semi-supervised learning.

Chapter 2

Semi-Supervised Learning and Approximation Methods

2.1 Gaussian Fields and Harmonic Functions

A semi-supervised learning scenario can be phrased as follows: data points $x_1, \dots, x_l \in \mathbb{R}^m$ are labeled with labels $f(1), \dots, f(l) \in \{0, 1\}$ and there are u unlabeled points x_{l+1}, \dots, x_n , where $n = l + u$. In (Zhu et al., 2003a), a fully-connected graph with all the points as nodes is considered and each edge is assigned a weight w_{ij} . The weight matrix W is symmetric and has non-negative elements. The case presented in the paper is that of exponential weights (although there can be other possibilities). Thus,

$$w_{ij} = \exp\left(-\sum_{d=1}^m \frac{(x_{id} - x_{jd})^2}{\sigma_d^2}\right)$$

where x_{id} is the d -th component of x_i . The σ_d are length-scale hyperparameters. The assumptions made are that the function f remains constrained at the labeled points, while at the unlabeled points it should minimise the

energy function

$$E(f) = \frac{1}{2} \sum_{i,j} w_{ij} (f(i) - f(j))^2$$

so that nearby points are encouraged to have similar labels. The function f is interpreted here as a Gaussian conditional probability distribution over the labels, with the first l being kept fixed. The minimising f is the mean of this distribution and satisfies $\Delta f = 0$ on the unlabeled points. Δ is the *combinatorial Laplacian* of the graph defined as

$$\Delta = D - W$$

where D is the diagonal matrix with the degrees

$$d_i = \sum_{j=1}^n w_{ij} \tag{2.1}$$

as entries. With $P = D^{-1}W$, it follows that $f = Pf$ on the unlabeled points. Written differently,

$$f(j) = \frac{1}{d_j} \sum_{i \sim j} w_{ij} f(i), \quad j = l+1, \dots, l+u \tag{2.2}$$

This relation implies (by a theorem of the harmonic functions) that $0 < f(i) < 1$ for unlabeled points or f is a constant.

Then the authors derive an expression for the function at the unlabeled points, denoted as a vector f_u , by splitting Δ in labeled and unlabeled parts:

$$f_u = (D_{uu} - W_{uu})^{-1} W_{ul} f_l = (I - P_{uu})^{-1} P_{ul} f_l \tag{2.3}$$

This equation classifies the unlabeled points. For learning the hyperparameters, an entropy minimisation is proposed. The entropy

$$H(f) = \frac{1}{u} = \sum_{i=l+1}^{l+u} H_i(f(i))$$

with

$$H_i(F(i)) = -f(i) \log(f(i)) - (1 - f(i)) \log(1 - f(i))$$

is minimised. Then a smoothing of the matrix P is applied in order to avoid a minimum of H at $\sigma_d = 0$. The smoothing replaces P with $\tilde{P} = \epsilon \mathcal{U} + (1 - \epsilon)P$, where $\mathcal{U}_{ij} = 1/(l + u)$. The gradient descent equations are then:

$$\begin{aligned} \frac{\partial H}{\partial \sigma_d} &= \frac{1}{u} \sum_{i=l+1}^{l+u} \log\left(\frac{1-f(i)}{f(i)}\right) \frac{\partial f(i)}{\partial \sigma_d} \\ \frac{\partial f_u}{\partial \sigma_d} &= (I - \tilde{P}_{uu})^{-1} \left(\frac{\partial \tilde{P}_{uu}}{\partial \sigma_d} f_u + \frac{\partial \tilde{P}_{ul}}{\partial \sigma_d} f_l \right) \\ \frac{\partial \tilde{P}}{\partial \sigma_d} &= (1 - \epsilon) \frac{\partial P}{\partial \sigma_d} \\ \frac{\partial p_{ij}}{\partial \sigma_d} &= \frac{\frac{\partial w_{ij}}{\partial \sigma_d} - p_{ij} \sum_{k=1}^{l+u} \frac{\partial w_{ik}}{\partial \sigma_d}}{\sum_{k=1}^{l+u} w_{ik}} \\ \frac{\partial w_{ij}}{\partial \sigma_d} &= 2w_{ij}(x_{id} - x_{jd})^2 / \sigma_d^3 \end{aligned} \tag{2.4}$$

2.2 Kd-trees

In the approximations of the weight matrix, it will be necessary to run fast searches for the nearest neighbours of points. Kd-trees are data structures designed for accelerating this kind of search. The aim is to arrange points $x_1, \dots, x_n \in \mathbb{R}^m$ on a binary tree, according to their relative positions, in order to dismiss as many points as possible during nearest neighbour search. Kd-trees were initially defined in (Bentley, 1975), (Friedman et al., 1977) and since then several variations have been proposed. The following description is based on (Moore, 1990) with notational changes.

kd-construct	
Input	$X = \{x_1, \dots, x_n\}, x_i \in \mathbb{R}^m$ <i>pivoting-method</i> (e.g. maximum variance, widest dimension etc.)
Output	kd (kd-tree of dimensionality m)
	<p>If $X = 0$, return the empty kd-tree.</p> <p>Call <i>pivoting-method</i>(X) to get i and s (the splitting dimension).</p> <p>Compute $X' = X \setminus \{x_i\}$.</p> <p>Compute $X_{left} = \{x_j \in X' : x_{js} \leq x_{is}\}$.</p> <p>Compute $X_{right} = \{x_j \in X' : x_{js} > x_{is}\}$.</p> <p>$kd_{left} := \mathbf{kd-construct}(X_{left}, \textit{pivoting-method})$</p> <p>$kd_{right} := \mathbf{kd-construct}(X_{right}, \textit{pivoting-method})$</p> <p>Return $kd := (x_i, i, s, kd_{left}, kd_{right})$.</p>

Table 2.1: Kd-tree construction algorithm

Each node kd in a kd-tree is associated with a value, an index (for indexing in the original set), a splitting dimension (according to which the tree is separated into left and right subtrees), a left and a right subtree:

$$(value, index, split, kd-left, kd-right).$$

There is also a function associated with a specific kd-tree which selects a splitting dimension at every node and a point (pivot). The algorithm for constructing a kd-tree is shown in Table 2.1. At any node, the set of remaining points is split to those which are smaller and those which are larger than the pivot point at the splitting dimension. Thus, the set of points is split into two hyperrectangles perpendicular to the splitting dimension, the pivot lying on the border.

To search for nearest neighbours on the kd-tree, the left and right subtrees at each node are tested for their closest distance from the target point

(see Table 2.2). The nearest hyperrectangle along the splitting dimension is always examined (2). For the furthest one, the closest point to the target is found and if it is within the nearest distance found so far, that subtree is examined recursively (4). In this case, the pivot point is also tested (3). This algorithm is called with an initial value of infinity for *max-dist-sqd* and the infinite hyperrectangle $[-\infty, +\infty]^m$ for *hr*.

The performance of the nearest neighbour search depends on how many calls of (3) are made. Ideally, only one subtree is examined at each node and the cost can be as low as $O(\log n)$. However, the distribution of the points can be such that the nearest found neighbours do not approach the target close enough along some dimension and a great number, close to $O(n)$, of hyperrectangles has to be examined. Pivoting strategies can compensate sometimes for undesirable distributions, since they determine the shape of the hyperrectangles formed. Two simple such strategies are the maximum variance strategy, i.e. selecting the dimension with the highest variance and the median point on it and the widest dimension strategy, which selects the dimension of widest spread of the points and the point closest to the middle as pivot.

2.3 Conjugate Gradients

Linear systems like (2.3), (2.4) can be solved efficiently with iterative methods if an approximate solution within some tolerance is acceptable. Otherwise, direct methods like Cholesky decomposition, which try to compute the exact solution, are available but cost more. Iterative methods are preferred with large and sparse coefficient matrices. Let a system $Ax = b$ be given. If A is a large, sparse matrix with M nonzero elements, the cost of an iterative method is $O(M)$ per iteration, while the total cost of a direct method is at least $O(n^2)$. The storage requirements are also $O(M)$, which can be helpful

kd-NN	
Input	kd (kd-tree of dimensionality m) $target \in \mathbb{R}^m$ $hr \in (\mathbb{R}^2)^m$ (hyperrectangle) $max-dist-sqd \in \mathbb{R}^+$
Output	$i \in \mathbb{N}$ (nearest neighbour index) $dist-sqd \in \mathbb{R}$
(1)	If kd is empty then return $dist-sqd := +\infty$. $x := kd_{value}$ $p := kd_{index}$ $s := kd_{split}$ Split hr into two hyperrectangles $left-hr$ and $right-hr$, at point x and dimension s . If $target_s \leq x_s$, $nearer-kd := kd_{left}$, $nearer-hr := left-hr$, $further-kd := kd_{right}$, $further-hr := right-hr$ Else $nearer-kd := kd_{right}$, $nearer-hr := right-hr$, $further-kd := kd_{left}$, $further-hr := left-hr$
(2)	$(i, dist-sqd) := kd\text{-NN}(nearer-kd, target, nearer-hr, max-dist-sqd)$ $max-dist-sqd := \min(max-dist-sqd, dist-sqd)$. Find closest point z to $target$ in $further-hr$: For $d = 1, \dots, m$ $z_d := \begin{cases} further-hr_{d,min} & \text{if } target_d \leq further-hr_{d,min} \\ target_d & \text{if } further-hr_{d,min} < target_d < further-hr_{d,max} \\ further-hr_{d,max} & \text{if } further-hr_{d,max} \leq target_d \end{cases}$ If $\ z - target\ ^2 < max-dist-sqd$,
(3)	$dsq := \ x - target\ ^2$ If $dsq < max-dist-sqd$, $i := p$ $dist-sqd := dsq$ $max-dist-sqd := dist-sqd$
(4)	$(temp-i, temp-dist-sqd) := kd\text{-NN}(further-kd, target, further-hr, max-dist-sqd)$ If $temp-dist-sqd < max-dist-sqd$, $i := temp-i$ $dist-sqd := temp-dist-sqd$ Return $i, dist-sqd$.

Table 2.2: Kd-tree nearest neighbour search

when there are few memory resources. One of the most standard iterative methods is the method of *Conjugate Gradients*. It was proposed in (Hestenes and Stiefel, 1952) and has spawned several variations. The standard method applies to symmetric positive definite coefficient matrices only. However, there are variations of conjugate gradients for other types of matrices too.

As described in (Shewchuk, 1994) and (Barrett et al., 1994), the simple conjugate gradient method works as follows. First, the problem solved can be seen as the minimisation of a quadratic form

$$\frac{1}{2}x^T Ax - b^T x + c$$

since differentiation, combined with symmetricity and positive definiteness, implies that there is a minimum when $Ax - b = 0$. A conjugate gradient descent thus solves the system by minimising iteratively the above quadratic function. The trick, however, is that the descent is performed (in theory) in exactly n steps, after the last of which it reaches the minimum. This is achieved by moving along n directions, $d_{(0)}, \dots, d_{(n-1)}$, which are *A-orthogonal*, or *conjugate*, to each other. This means that $d_{(i)}^T A d_{(j)} = 0$, for every $i \neq j$. Moreover, each time, the search minimises the function along the search direction $d_{(i)}$. This set of n *A-orthogonal* directions could be constructed in various ways (Gram-Schmidt orthogonalisation, for instance), but one construction is particularly suitable. This construction ensures conjugacy among all the $d_{(i)}$ while keeping only the current and previous estimates in memory. At each iteration, it keeps the estimates $x_{(i+1)}, x_{(i)}$, the residuals $r_{(i+1)}, r_{(i)} = b - Ax_{(i)}$ and the directions $d_{(i+1)}, d_{(i)}$. The update equations are:

$$d_{(0)} = r_{(0)} = b - Ax_{(0)}$$

$$\alpha_{(i)} = \frac{r_{(i)}^T r_{(i)}}{d_{(i)}^T A d_{(i)}}$$

$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} d_{(i)}$$

$$\begin{aligned}
r_{(i+1)} &= r_{(i)} - \alpha_{(i)} A d_{(i)} \\
\beta_{(i+1)} &= \frac{r_{(i+1)}^T r_{(i+1)}}{r_{(i)}^T r_{(i)}} \\
d_{(i+1)} &= r_{(i+1)} + \beta_{(i+1)} d_{(i)}
\end{aligned}$$

The main attraction of this method is that storage requirements are limited and only vector-vector and matrix-vector products are computed. Hence the $O(M)$ complexity.

Although theoretically the process terminates at the n -th iteration at most, in practice there is round-off error which leads to loss of accuracy and loss of orthogonalisation of the search directions. Moreover, with large matrices convergence in $O(n)$ iterations is not acceptable. So, the method is used as an approximation of the actual solution and is terminated when a satisfactory tolerance level is reached. Under certain assumptions, an upper bound for the number of iterations required to reach a certain tolerance is given by $O(\sqrt{\kappa})$, where κ is the spectral condition number, equal to the ratio of the largest to the smallest eigenvalue (see (Shewchuk, 1994)). This implies an upper bound for time complexity of $O(M\sqrt{\kappa})$.

Chapter 3

Weight Matrix Construction

3.1 Weighting Schemes

As described in Section 2.1, the weight matrix W is an $n \times n$ symmetric matrix representing the structure of the nodes in the graph G . It is desirable that large weights should correspond to pairs of points which are “close” to each other and small weights to “distant” points. Usually, the weights w_{ij} are some decreasing function of the Euclidean distances d_{ij} . In (Zhu et al., 2003a) and (Zhu et al., 2003b), the authors consider the following possibilities:

- Unweighted kNN graph: $w_{ij} = 1$ if i is one of the k nearest neighbours of j , or vice versa.
- Unweighted ϵ NN graph: $w_{ij} = 1$ if $d_{ij} \leq \epsilon$.
- tanh-weighted graph: $w_{ij} = (\tanh(\alpha_1(d_{ij} - \alpha_2)) + 1)/2$, where α_1, α_2 are hyperparameters controlling the sigmoid decrease in weights.
- exp-weighted graph: $w_{ij} = \exp(-\sum_{d=1}^m \frac{(x_{id}-x_{jd})^2}{\sigma_d^2})$, where x_{id} is the component of x_i along the d -th dimension and the σ_d are length scale

hyperparameters controlling the decay of the weights with respect to distance along each dimension.

In general, the exact computation of all the w_{ij} has $O(n^2)$ time complexity. However, in the cases of kNN weights and ϵ NN weights, kd-tree algorithms can be used (Moore, 1990, 6.7.4, 6.7.1). The k nearest neighbours of a node can be found with a straightforward modification of the algorithm in Table 2.2. The list of the k (or less) nearest neighbours found so far is an input passed to the recursive calls (??, ??) and updated in ??. The list is kept sorted so that the maximum distance is updated to the distance from the k -th nearest neighbour (??). For the ϵ NN graph, a range search is needed, that is finding all the nearest neighbours within ϵ . The maximum distance is set to ϵ and does not change, whereas any neighbours found within this distance are appended to a list and returned as output. Kd-trees can improve the average performance under certain conditions (see Section 2.2). If the kd-tree constructed is reasonably balanced, the hyperrectangles are close to hypercubic and the dimensionality is low, then the cost of finding a nearest neighbour will be $O(\log n)$ on average. This would result in $O(kn \log n)$ cost for the kNN case and $O(e(\epsilon) \log n)$ cost for the ϵ NN case, where $e(\epsilon)$ is the number of edges on the graph with distance $d_{ij} \leq \epsilon$.

In the continuous schemes, the weight computation may also be sped up by considering only large weights and setting all other weights to zero. If the weights relate to the distances through a decreasing function, this approximation is equivalent to a kNN or a range search on the distances. The only difference is that now the distances of the nearest neighbours have to be stored, since the nonzero weights can be different from 1. This method applies to the exp-weighted case too, by normalising first the points with the length scale parameters: $x'_{id} = x_{id}/\sigma_d$, $d = 1, \dots, m$. The intuition is that non-neighbouring points should have little contribution to the label of a point, as shown by the harmonic property (2.2). Therefore, considering only

the nearest neighbours should not drastically change the solution for f .

3.2 Kd-trees in High Dimensions

When using kd-trees for finding nearest neighbours, an important consideration is the so-called “curse of dimensionality”. While kd-trees offer great performance gains when the dimensionality m is small (less than 10-20), in higher dimensions they perform no better than the naive search algorithm, i.e. $O(n)$ for nearest neighbour search. The main reason, as noted in (Weber et al., 1998), is that most of the time the distance from the nearest neighbour is larger than the range of the data space, when m is not small. For example, for data points uniformly and independently distributed inside a hypercube, the average nearest neighbour distance increases with m and is already larger than the edge of the hypercube after $m = 30$. Then, a kd-tree search with a radius larger than the range of the data space has linear cost because it implies that all the hyperrectangles need to be examined for nearest neighbours.

We have to address this issue because the dimensionality of the data sets we are interested in is at least in the hundreds. One way would be to project the data on just a small number of dimensions, in other words to find the few most significant features. Unfortunately, it is often the case that there is important information in many of the components. For example, consider the image data from the handwritten digits ‘1-2’ data set used in (Zhu et al., 2003a) (see Figure 3.1). Each data point is a 256-dimensional vector (a 16×16 image), the values along each dimension range from 0 to 255 and the points correspond to digits ‘1’ or ‘2’ (1100 points from each class). It is apparent that most parts of each image contain some information relevant to the classification task. This can also be seen in the variances (Figure 3.2), which are high in more than half the pixels.

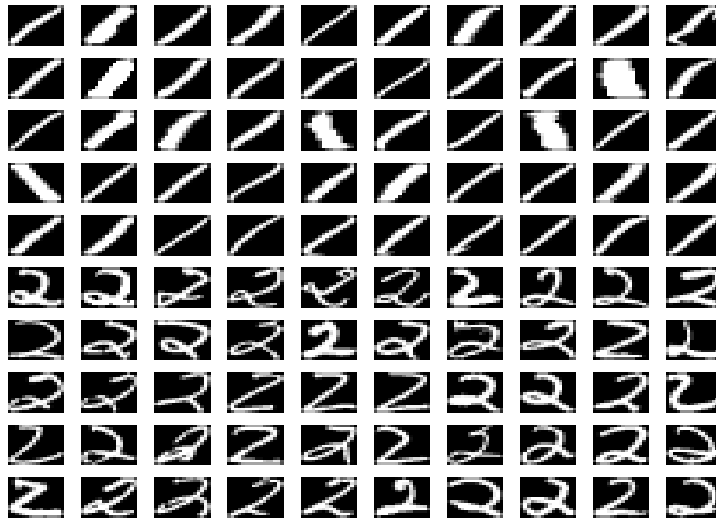


Figure 3.1: Sample from the '1-2' data set.

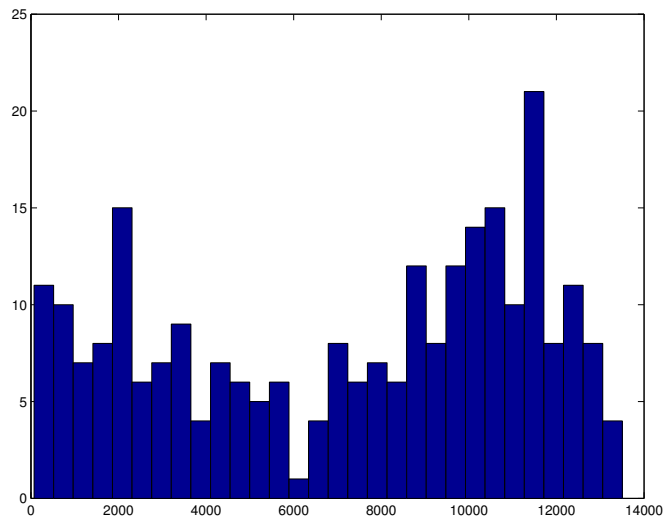


Figure 3.2: Histogram of variances of values at every pixel.

Thus, it is necessary to “mix” values from all the pixels into few components without losing much information. A common approach is *Principal Component Analysis (PCA)* (see e.g. (Bishop, 1995)). In brief, the covariance matrix of the data points (after subtraction of the mean) is computed and then the eigenvalues and eigenvectors of the covariance matrix are found. The largest eigenvalues and the corresponding eigenvectors are kept and these vectors form the basis of the new low-dimensional space on which the data points are projected:

$$\sum_{i=1}^n (x_i - \bar{x})(x_i - \bar{x})^T = V \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \lambda_m \end{pmatrix} V^T \quad (3.1)$$

where $\lambda_1 \geq \dots \geq \lambda_m$ are the eigenvalues of $\sum_{i=1}^n (x_i - \bar{x})(x_i - \bar{x})^T$ and V is orthogonal.

One advantage of PCA is that the largest eigenvalues distinguish the directions in the transformed space $(x - \bar{x})^T V$ having the highest variance. In practice, a small number of the first columns of V usually approximates most of the variance in the data. This can be beneficial to the performance of kd-tree based algorithms, because the range of values along these eigenvectors can be higher than the average nearest neighbour distance. This is demonstrated in Figures 3.3 , 3.4, where the ‘1-2’ data set was used.

Therefore, for small PCA dimensionality the search should be $O(\log n)$ on average and for larger values linear. In Figure 3.5, the computational cost was estimated using the number of distance computations (Table 2.2, 3) and was measured against sample size. The data set used was the handwritten digits ‘0’ to ‘9’ data set from (Zhu et al., 2003a). Each time, a random sample was selected and 500 random nearest neighbour searches were performed in this sample. Their average cost was recorded as the cost estimate. In 12 dimensions the cost exhibits logarithmic behaviour, whereas in 20 dimensions

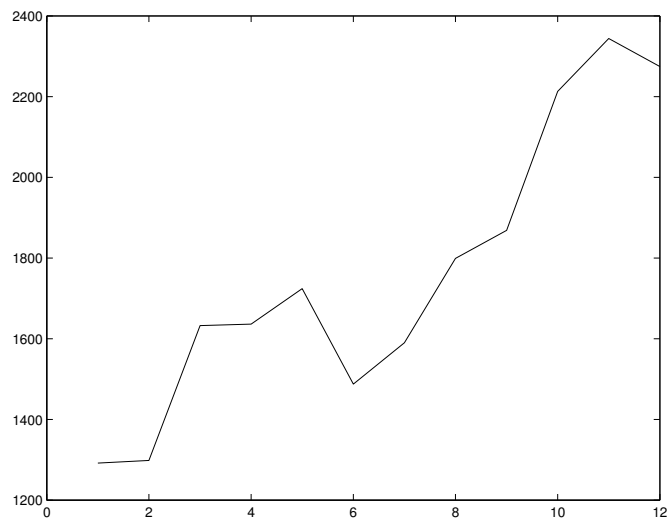


Figure 3.3: Range of values along the 12 principal components.

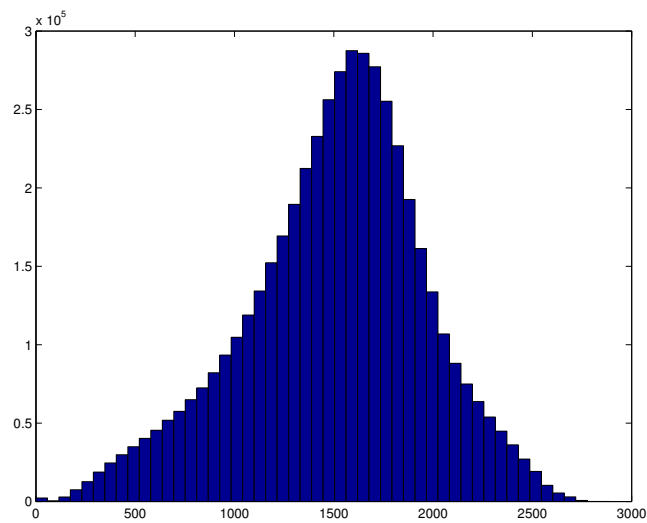


Figure 3.4: Histogram of distances in the subspace of the 12 principal components.

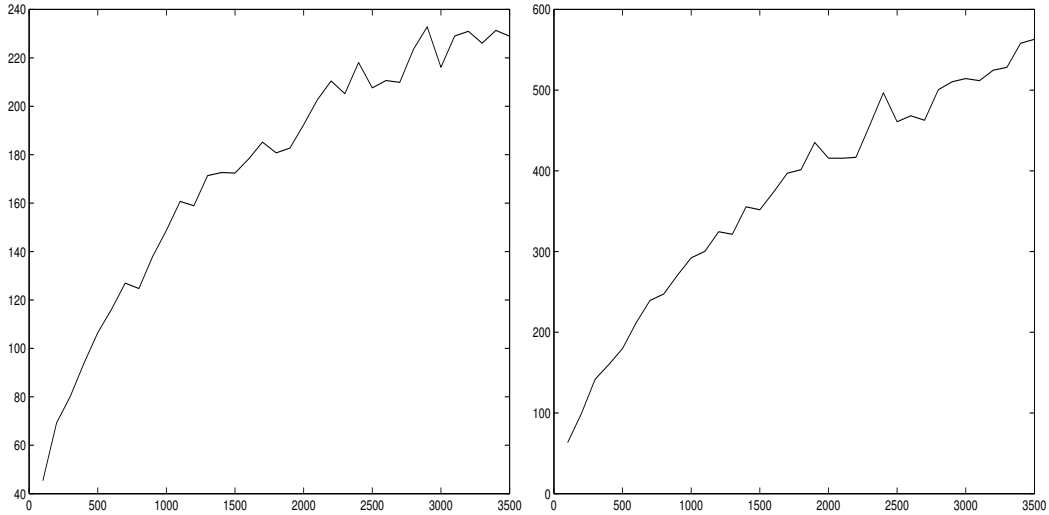


Figure 3.5: Computational cost of kd-NN versus sample size with a data set projected on 12 (left) and 20 (right) principal components. Maximum variance was used for pivot selection.

it starts to approach a linear function, indicating that most of the nearest neighbour distances exceed the range.

Since the PCA dimension has to be limited, it is not possible to get a good approximation of the distances between points in the data set. From (3.1) it is evident that PCA before eigenvector selection is a distance-preserving transformation, so eventually a distance between a pair of points will be smaller than the original one. Some of the rejected eigenvectors may have a non-negligible contribution to the distance, which implies that the approximation is rough. A measure of how well μ principal components describe the data is given by the fraction of the variance of the data along them, which equals $\frac{\sum_{i=1}^{\mu} \lambda_i}{\sum_{i=1}^m \lambda_i}$. With the ‘1-2’ data set, for instance, the fraction of the variance along the 12 principal components is just 0.67 and with 30 principal components it is 0.84. This is reflected in a significant difference between nearest neighbours in the projected space and actual ones. The fraction of

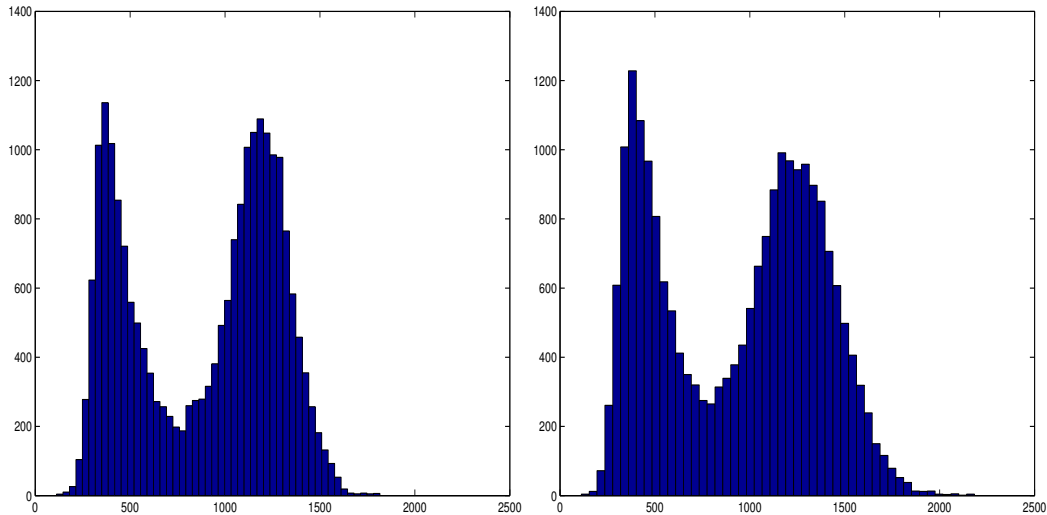


Figure 3.6: Histogram of the 10 nearest neighbour distances for all points. Nearest neighbours were selected from the original space (left) and from the space of the 12 principal components (right).

the actual 10 nearest neighbours found by searching the 10 nearest neighbours in 12 dimensions is 0.62. This ratio increases as k increases but so does computational cost. However, the distribution of (original) distances of the k nearest neighbours in the projected space and that in the original space do not differ a lot (see Figure 3.6). The reason is that there is enough density in the data so that in the neighbourhood of a point distances tend to be similar. Moreover, the class in which nearest neighbours belong tends to be consistent in the two cases. In the above example, 99% of the nearest neighbours of a point have the same label in both cases. Finally, it should be noted that as the number of classes increases k should also increase in order to maintain the same accuracy.

In addition to the computational cost incurred by kd-tree search, the cost of PCA should also be considered. In total, the improvement in kd-tree performance exceeds the additional preprocessing cost. Computing the

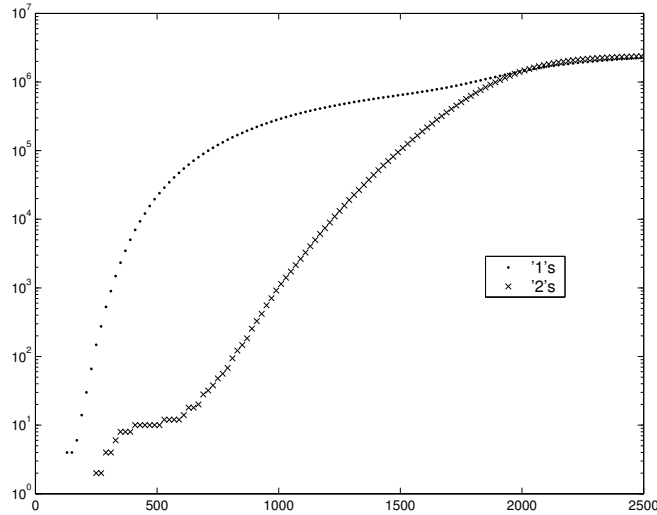


Figure 3.7: Logarithm of the number of ϵ nearest neighbours of images of ‘1’ and of images of ‘2’, versus ϵ .

covariance matrix in (3.1) takes $O(nm^2)$. Eigenvalue decomposition takes $O(m^3)$, but can be done in $O(m^2)$ using iterative methods, since we only need the first few eigenvectors. These methods, like conjugate gradients, rely on Krylov subspaces for orthogonalisation, the most common of them being Lanczos-Arnoldi (see e.g. (Parlett, 1998)). In addition, m may be reduced by smoothing or another technique if it is too large. Thus, the contribution of PCA to the total cost could be large with larger dimensionality but there are ways it can be reduced.

3.3 ϵ NN Search

The previous discussion applies to ϵ NN search as well as to kNN search and the behaviour inside an ϵ -neighbourhood of a point should be similar to that in a k -neighbourhood. There is an important difference, however: the number

of points inside an ϵ -neighbourhood may differ widely across the data set. So, for instance, the ϵ -neighbourhood of an image of the digit ‘1’ is more dense than that of an image of digit ‘2’, since distances between ‘1’s tend to be smaller than distances between ‘2’s. As Figure 3.7 shows, the population inside ϵ -neighbourhoods of ‘1’s can be two orders of magnitude larger than that inside ϵ -neighbourhoods of ‘2’s. At the value of ϵ for which there are on average 2 ϵ -neighbours of ‘2’s, there are on average 310 ϵ -neighbours of ‘1’s. As can be expected, applying PCA does not alter this behaviour. It is consistent with the observations in (Zhu et al., 2003b):

- (a) The accuracy of learning shows fluctuations, probably because the ϵ -neighbourhood of certain points (‘2’s in the above example) is sparse and because of concentrated clusters of the same class.
- (b) The number of edges under the optimal ϵ can be much larger than that of kNN graphs, which can be explained by the large density in certain classes.

Thus, there is a tradeoff in the choice of ϵ . Smaller values may produce too few nearest neighbours for sparsely distributed classes whereas larger values may produce a huge number of edges for densely distributed classes. With large data sets, the quality of learning has to suffer and kNN graphs seem a more promising approach.

3.4 Effect of the Pivoting Method

As described in Section 2.2, the method for choosing the pivot point during kd-tree construction may have a significant effect on the speed of nearest neighbour search. Certain methods are better suited for certain distributions than others. Therefore, some preliminary investigation may be needed in

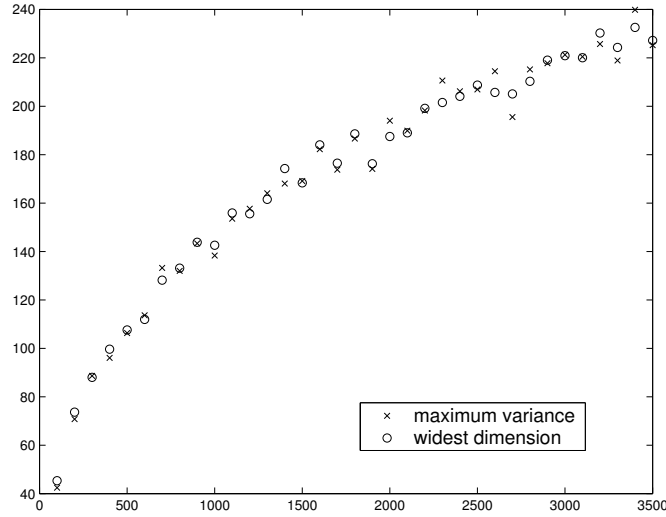


Figure 3.8: Computational cost of kd-NN versus sample size, on 12 principal components using two different pivoting strategies.

order to find a good pivoting strategy to use. Figure 3.8 shows the same experiment as in Figure 3.5 run using both the maximum variance method and the widest dimension method. It seems that there is not a real difference in performance, which is consistent with the distribution of the data after PCA. This distribution should be more uniformly spread and less likely to have large empty areas than the original one. Consequently, there will not be a great number of disproportionate hyperrectangles, which are more likely to be inspected by the kd-tree search algorithm. The shape of the tree produced, which can be a problem with the widest dimension method, was fairly balanced. As expected, the tree produced by maximum variance was almost perfectly balanced. Hereafter, only maximum variance will be used in the experiments performed and the pivoting method will only have marginal effect.

3.5 Implementation

Table 3.1 summarises the algorithm for constructing a sparse weight matrix. Tables 3.2, 3.3 describe kNN and ϵ NN search. In kNN search, a list of candidate neighbours is passed as an argument and can be modified when a nearer point is found. The list is kept sorted in increasing order with respect to distance, so that the radius of the search is always the distance from the k -th element (or $+\infty$ if there are less than k elements). Whenever a nearer point is found, it is inserted in place according to its distance. In ϵ NN search, the radius of the search remains fixed and equal to ϵ^2 . Any points found inside this distance are added to the list of nearest neighbours. Finally, the returned nearest neighbour distances are in PCA dimensionality, because the original distances can be computed more efficiently on X .

construct-weights	
Input	$X = \{x_1, \dots, x_n\}, x_i \in \mathbb{R}^m$ <i>pivoting-method</i> (e.g. maximum variance, widest dimension) <i>weighting-method</i> (kNN or ϵ NN) <i>hparam</i> (k or ϵ) $\mu \in \mathbb{N}$ (number of eigenvectors for PCA)
Output	W ($n \times n$ symmetric matrix with positive entries)
PCA	Compute $\sum_{i=1}^n (x_i - \bar{x})(x_i - \bar{x})^T$ and its first μ eigenvectors v_1, \dots, v_μ . Compute $\mathcal{X} = \{(x_i^T v_1, \dots, x_i^T v_\mu)\}$.
Kd-tree construction	$kd := \text{kd-construct}(\mathcal{X}, \textit{pivoting-method})$ ⁽¹⁾
Kd-tree search	For $i = 1, \dots, n$ If <i>weighting-method</i> is kNN, $nn\text{-list}_i := \text{kNN}(kd, X, x_i, \chi_i, \{\}, \mathbb{R}^m, \textit{hparam})$ ⁽²⁾ Else if <i>weighting-method</i> is ϵ NN, $nn\text{-list}_i := \epsilon\text{NN}(kd, X, x_i, \chi_i, \mathbb{R}^m, \textit{hparam}^2)$ ⁽³⁾ For $i = 1, \dots, n$ For each $(j, \textit{dist-sqd})$ in $nn\text{-list}_i$, set $w_{ij} := w_{ji} := \textit{dist-sqd}$. Return sparse matrix W with entries w_{ij} .

Table 3.1: Weight matrix construction algorithm

¹ see Table 2.1

² see Table 3.2

³ see Table 3.3

kNN	
Input	kd (kd-tree of dimensionality m) $target \in \mathbb{R}^\mu$ (target after projection) $initial-nn-list$ (list of indexes and squared distances $(j, dist-sqd)$) $hr \in (\mathbb{R}^2)^m$ (hyperrectangle) $k \in \mathbb{N}$
Output	$nn-list$ (list of indexes and squared distances of nearest neighbours $(j, dist-sqd)$)
<p>If kd is empty then return $initial-nn-list$.</p> <p>$x := kd_{value}$ $p := kd_{index}$ $s := kd_{split}$</p> <p>Split hr into two hyperrectangles $left-hr$ and $right-hr$, at point x and dimension s.</p> <p>If $target_s \leq x_s$, $nearer-kd := kd_{left}$, $nearer-hr := left-hr$, $further-kd := kd_{right}$, $further-hr := right-hr$</p> <p>Else $nearer-kd := kd_{right}$, $nearer-hr := right-hr$, $further-kd := kd_{left}$, $further-hr := left-hr$</p> <p>$nn-list := \text{kNN}(nearer-kd, target, initial-nn-list, nearer-hr, k)$ $max-dist-sqd := \text{last}(nn-list)_{dist-sqd}$ (or $+\infty$ if $nn-list$ is empty)</p> <p>Find closest point z to $target$ in $further-hr$:</p> <p>For $d = 1, \dots, m$</p> $z_d := \begin{cases} further-hr_{d,min} & \text{if } target_d \leq further-hr_{d,min} \\ target_d & \text{if } further-hr_{d,min} < target_d < further-hr_{d,max} \\ further-hr_{d,max} & \text{if } further-hr_{d,max} \leq target_d \end{cases}$ <p>If $\ z - target\ ^2 < max-dist-sqd$, $dsq := \ x - target\ ^2$ If $dsq < max-dist-sqd$, If $\text{length}(nn-list) = k$, remove the last element. Insert (p, dsq) into $nn-list$ by comparison with the values of $dist-sqd$. $nn-list := \text{kNN}(further-kd, target, nn-list, further-hr, k)$</p> <p>Return $nn-list$.</p>	

Table 3.2: k nearest neighbours algorithm

ϵ NN	
Input	kd (kd-tree of dimensionality m) $target \in \mathbb{R}^m$ $hr \in (\mathbb{R}^2)^m$ (hyperrectangle) $\epsilon\text{-sqd} \in \mathbb{R}^+$
Output	$nn\text{-list}$ (list of indexes and squared distances of nearest neighbours $(j, dist\text{-sqd})$)
	<p>If kd is empty then return an empty list.</p> <p>$x := kd_{value}$ $p := kd_{index}$ $s := kd_{split}$</p> <p>Split hr into two hyperrectangles $left\text{-hr}$ and $right\text{-hr}$, at point x and dimension s.</p> <p>If $target_s \leq x_s$, $nearer\text{-kd} := kd_{left}$, $nearer\text{-hr} := left\text{-hr}$, $further\text{-kd} := kd_{right}$, $further\text{-hr} := right\text{-hr}$</p> <p>Else $nearer\text{-kd} := kd_{right}$, $nearer\text{-hr} := right\text{-hr}$, $further\text{-kd} := kd_{left}$, $further\text{-hr} := left\text{-hr}$</p> <p>$nn\text{-list} := \epsilon\text{NN}(nearer\text{-kd}, target, nearer\text{-hr}, \epsilon\text{-sqd})$</p> <p>Find closest point z to $target$ in $further\text{-hr}$:</p> <p>For $d = 1, \dots, m$</p> $z_d := \begin{cases} further\text{-hr}_{d,min} & \text{if } target_d \leq further\text{-hr}_{d,min} \\ target_d & \text{if } further\text{-hr}_{d,min} < target_d < further\text{-hr}_{d,max} \\ further\text{-hr}_{d,max} & \text{if } further\text{-hr}_{d,max} \leq target_d \end{cases}$ <p>If $\ z - target\ ^2 < \epsilon\text{-sqd}$, $dsq := \ x - target\ ^2$ If $dsq < \epsilon\text{-sqd}$, Add (p, dsq) into $nn\text{-list}$. $temp\text{-list} := \epsilon\text{NN}(further\text{-kd}, target, further\text{-hr}, \epsilon\text{-sqd})$ Append $temp\text{-list}$ to $nn\text{-list}$.</p> <p>Return $nn\text{-list}$.</p>

Table 3.3: ϵ nearest neighbours algorithm

Chapter 4

Solving the Harmonic Equation

4.1 Sparse Linear System Solution

The solution of equation (2.3) can be rewritten as the linear system

$$\Delta_{uu}f_u = Wulf_l \tag{4.1}$$

where $\Delta = D - W$. It should be noted that Δ_{uu} is not always strictly positive definite. This can be seen from the expansion of $x^T \Delta_{uu} x$ and (2.1). Singularity of the matrix occurs when there are isolated subgraphs of unlabeled points, a situation not desirable which implies that some of the structure in the graph is not being exploited. With k or ϵ large enough this is less likely to happen. In any case, as suggested in (Smola and Kondor, 2003) and (Zhu et al., 2003b), a possible singularity can be removed by regularising the eigenvalues through the function

$$r(\lambda) = \lambda + 1/\sigma^2.$$

This is equivalent to replacing the Laplacian with the *regularised Laplacian* $\tilde{\Delta} = \Delta + \mathbf{I}/\sigma^2$.

A linear system like (4.1) can be solved efficiently when Δ_{uu} is sparse with $M = O(n)$ non-zero elements. A Laplacian matrix generated as in Chapter

3 can be more than 99.5% sparse in the case of kNN or about 98% sparse in the case of ϵ NN. There are several direct and iterative methods which can be applied to (4.1) and it has to be investigated which one is best suited to the type of matrix and the requirements of the problem at hand.

4.2 Iterative Methods

Iterative methods tackle the solution of a linear system with successive approximations and can be either *stationary* or *non-stationary*, depending on whether the iteration coefficients stay the same or not. *Jacobi*, *Gauss-Seidel* and *successive overrelaxation* are common stationary methods. Non-stationary methods include *conjugate gradients* and their many variants, *Newton-Raphson*, *Chebyshev iteration* etc. For a good review of iterative methods see (Barrett et al., 1994).

Conjugate gradients, in particular, have been widely used, especially in problems where the matrix is very large and sparse. The appeal of these methods versus direct methods or Newton-Raphson is that they require only matrix-vector products at each iteration. Their disadvantages are a possibly larger number of iterations needed for convergence and less stability. Because of round-off errors in the computations, the desired amount of relative tolerance has to be specified as a stopping condition. The stopping criterion can vary but usually involves the relative residual $\|Ax_{(i)} - b\|/\|b\|$.

Complexity depends on the condition number κ of the matrix, which affects the number of iterations needed for achieving a given tolerance. Recall from Section 2.3 the theoretical upper bound of $O(M\sqrt{\kappa})$ for time and $O(M)$ for space. However, it has been observed that conjugate gradients perform well even when the condition number is large but the extremal eigenvalues of the matrix are well separated (see (van der Sluis and van der Vorst, 1986), (Concus et al., 1976)). The reason is that the method concentrates quickly

on the inner eigenvalues, which is a problem with smaller condition number, and so the rate of convergence increases with each iteration (*superlinear* convergence).

There are several methods in the family of conjugate gradients and which to use depends on the specifics of the problem at hand. In our case, the main concerns are:

- Storage requirements are less important than CPU time. The weight and Laplacian matrices after sparsification will not consume large amounts of memory space, so any improvements in time complexity should be applied. Moreover, the most time-consuming task is learning the weights, which requires m solutions of equation (??).
- The main matrix involved is positive definite (after regularisation) and symmetric. However, equation (4.1) can also be written in terms of $P = D^{-1}W$ as

$$(I - P_{uu})f_u = P_{ul}f_l, \quad (4.2)$$

which involves the positive semi-definite but generally nonsymmetric matrix $I - P_{uu}$. Thus, both symmetric and nonsymmetric methods should be considered.

- Stability can be important because the unlabeled Laplacian may be close to singular.

To compare the different iterative methods, I applied each one to the systems $\Delta_{uu}x_i = b_i$ and $(I - P_{uu})x_i = b_i$, where x_1, \dots, x_{50} were selected randomly and uniformly from the hypercube $[0, 1]^m$ and b_1, \dots, b_{50} were computed from the x_i s. The average time cost, relative residual and error from the actual x_i were recorded. The Laplacian was obtained from the ‘0-9’ data set, which has 4000 points, and the number of unlabeled points varied between 2000 and 3900. The results are shown in Figures 4.1-4.3.

For the execution of the methods, the corresponding Matlab procedures were called (see (Matlab Documentation,)). For symmetric matrices the applicable methods are *Conjugate Gradient (CG)*, *Minimum Residual (MINRES)* and *Symmetric LQ (SYMMLQ)*, whereas for nonsymmetric matrices *Generalised Minimal Residual (GMRES)*, *Biconjugate Gradient (BICG)*, *Biconjugate Gradient Stabilised (BICGSTAB)*, *Quasi-Minimal Residual (QMR)*, *Conjugate Gradient Squared (CGS)* are more appropriate. As mentioned in (Barrett et al., 1994), CG is the first of these methods to have been proposed and applies to symmetric positive definite matrices only. MINRES and SYMMLQ are variants designed for indefinite systems, where CG may have unstable behaviour. SYMMLQ yields the same results as CG in the positive definite case. GMRES is an extension of MINRES for the nonsymmetric case. Without symmetry, however, the whole orthogonal basis constructed needs to be stored in memory. Therefore, the computational cost is higher per iteration and this method has to be restarted every time a number of iterations, specified by the user, is reached. This restart parameter affects convergence and is often hard to choose because there is no clear rule as to its most appropriate values. BICG is similar to CG but uses two mutually orthogonal sequences. It is appropriate for nonsingular matrices and in the symmetric case produces the same results as CG in more time. Its convergence to the solution is not always regular, so one of the following three variants is often preferable. QMR has better convergence properties by using least squares minimisation. Its Matlab implementation gives the same results as MINRES for symmetric matrices, but in more time. CGS, which effectively applies conjugation twice, in many cases enhances the speed of convergence but may be irregular. Finally, BICGSTAB combines BICG and GMRES to obtain smoother convergence than CGS in many cases.

To gain some insight into the experimental results, we can examine the condition number of Δ_{uu} and $I - P_{uu}$ at the same values of u (Figure 4.4).

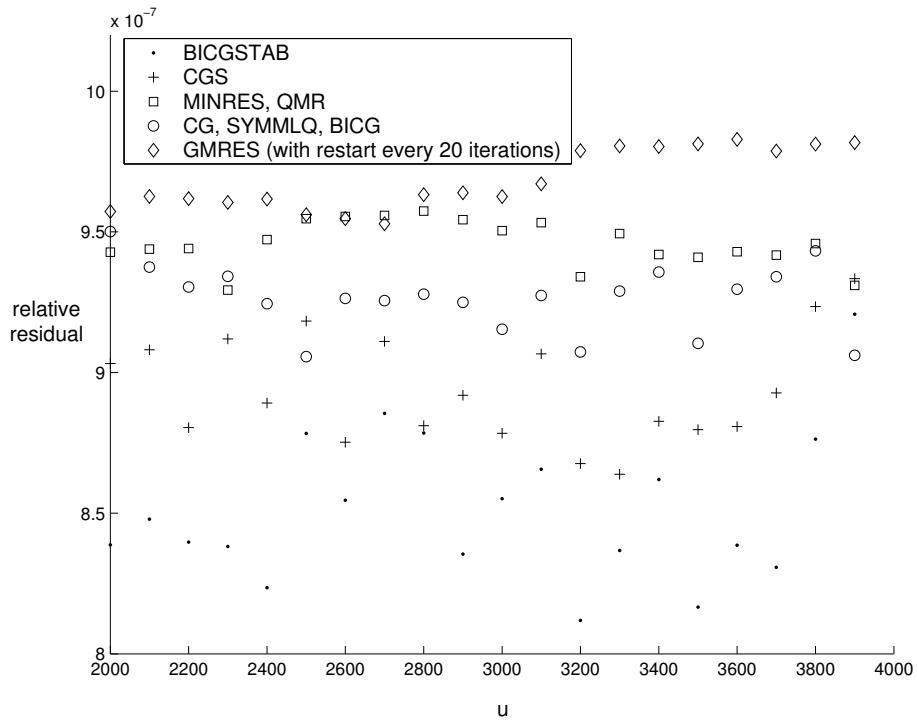
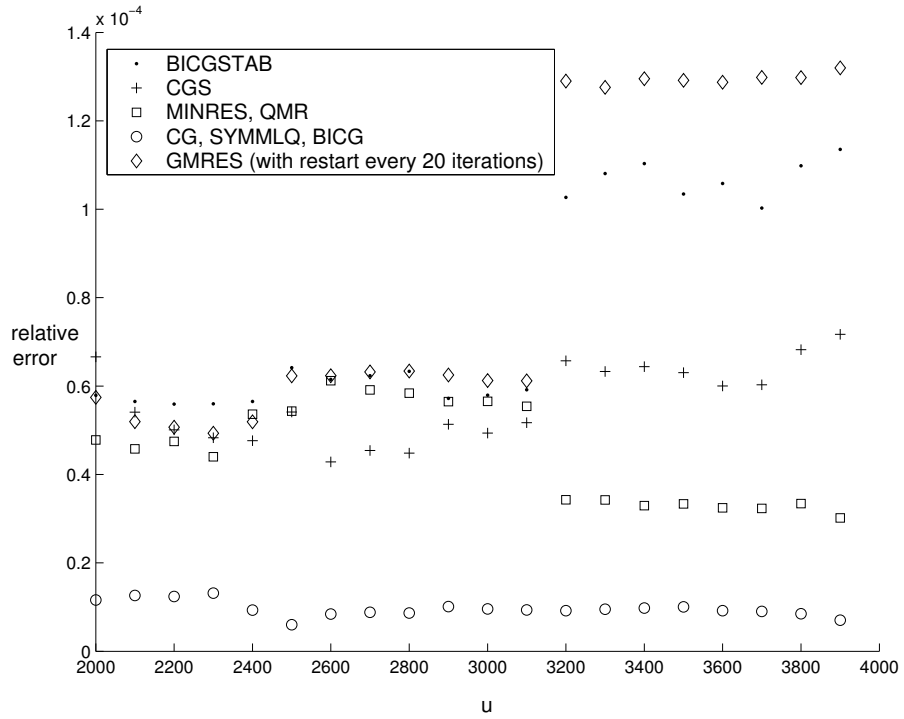


Figure 4.1: Accuracy $\frac{\|\tilde{x}-x\|}{\|x\|}$ (top) and relative residual (bottom) for various iterative methods using Δ_{uu} as coefficient matrix. The stopping criterion is a 10^{-6} tolerance for the relative residual. In all cases, the algorithms converged within this tolerance.

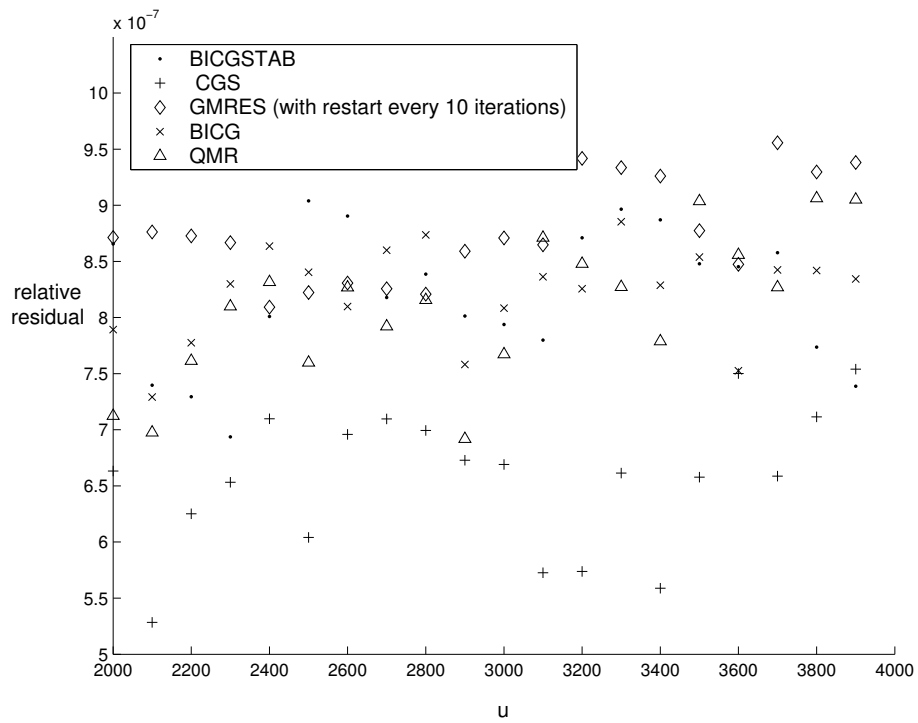
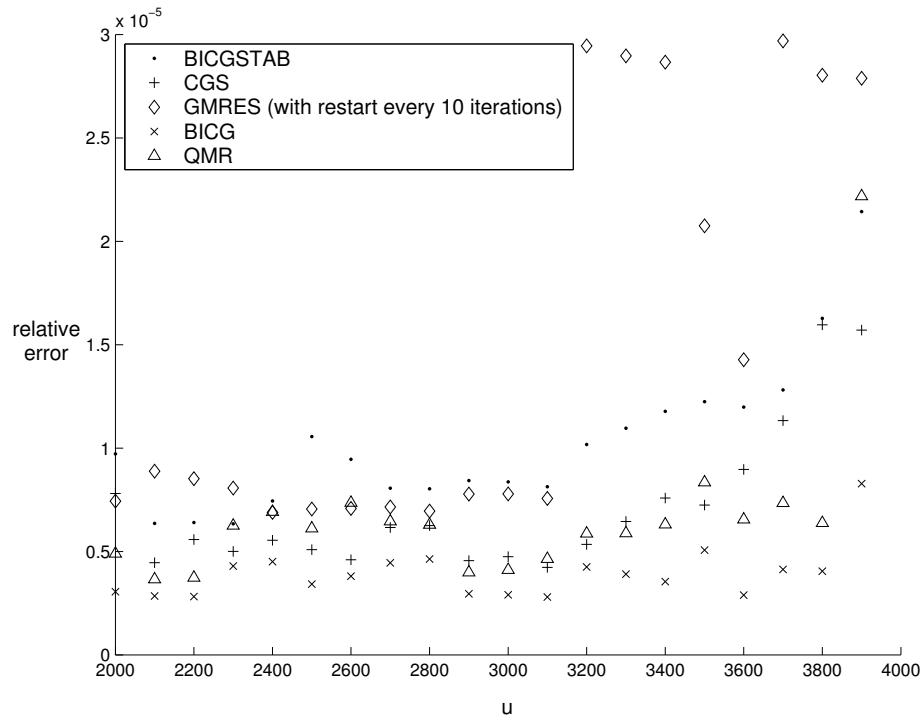


Figure 4.2: Accuracy (top) and relative residual (bottom) for various iterative methods using $I - P_{uu}$ as coefficient matrix.

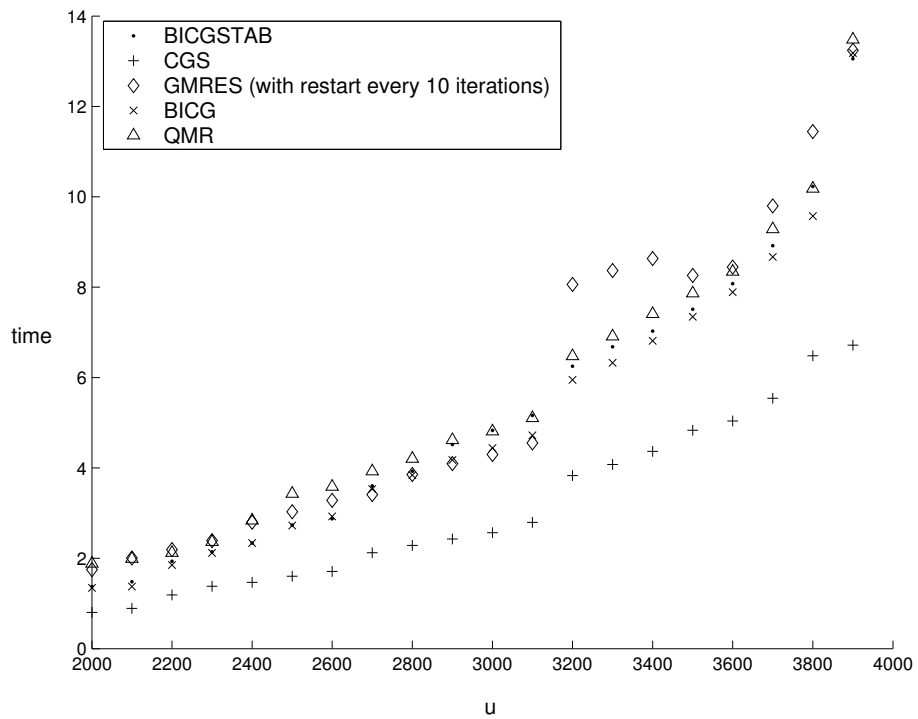
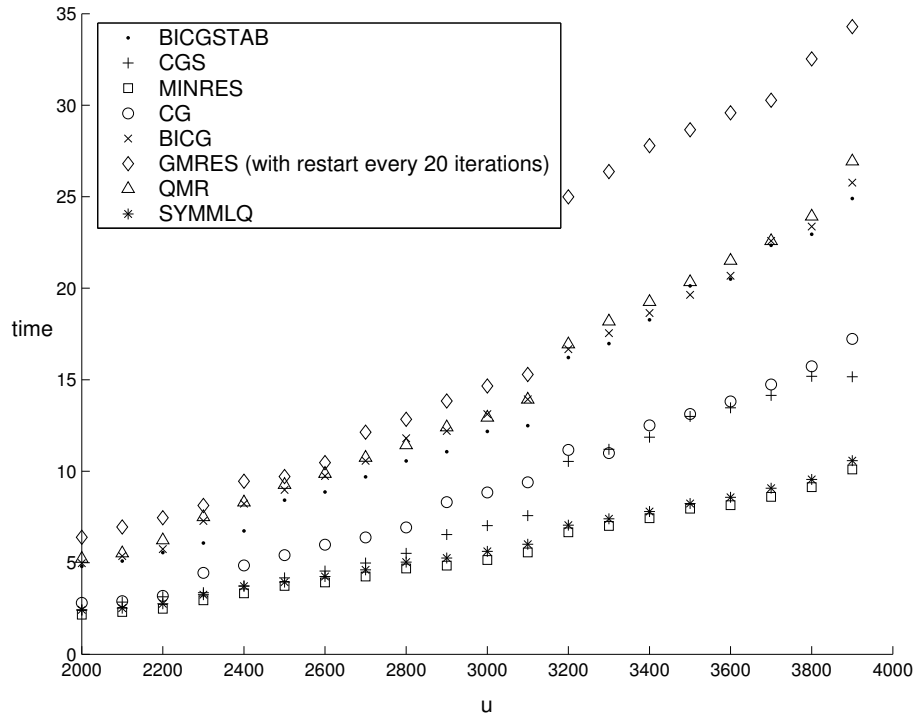


Figure 4.3: Required time for various iterative methods using Δ_{uu} (top) and $I - P_{uu}$ (bottom) as coefficient matrix.

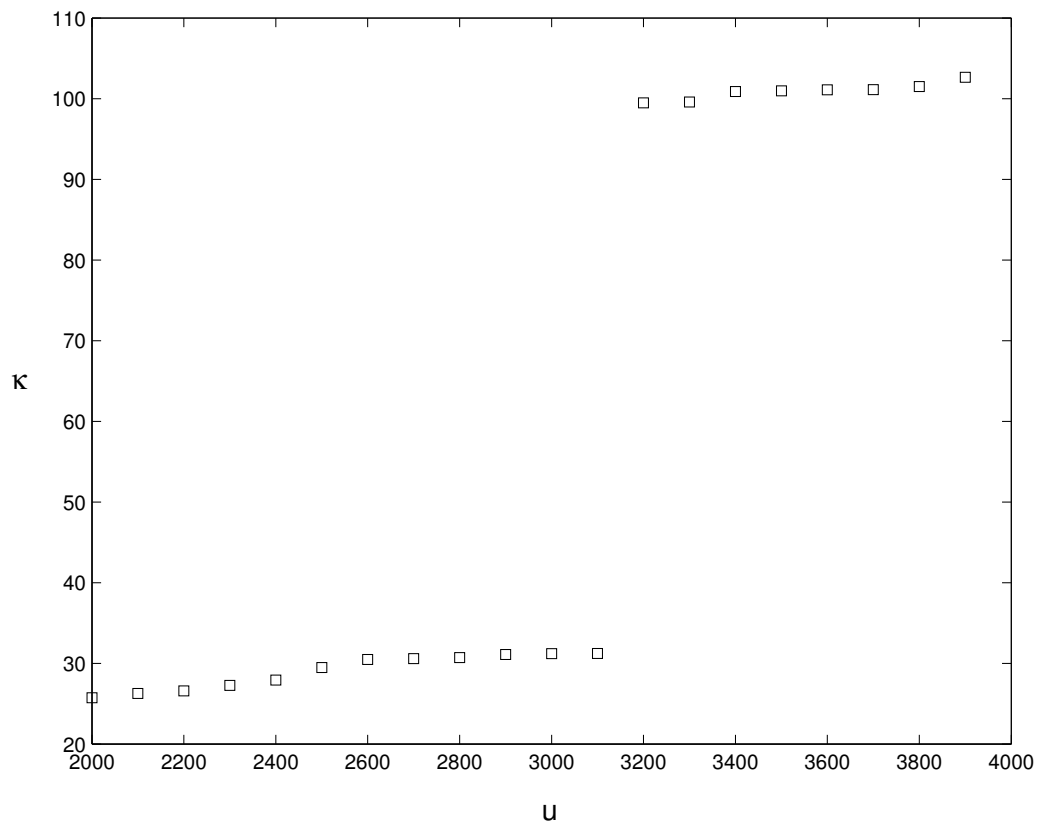
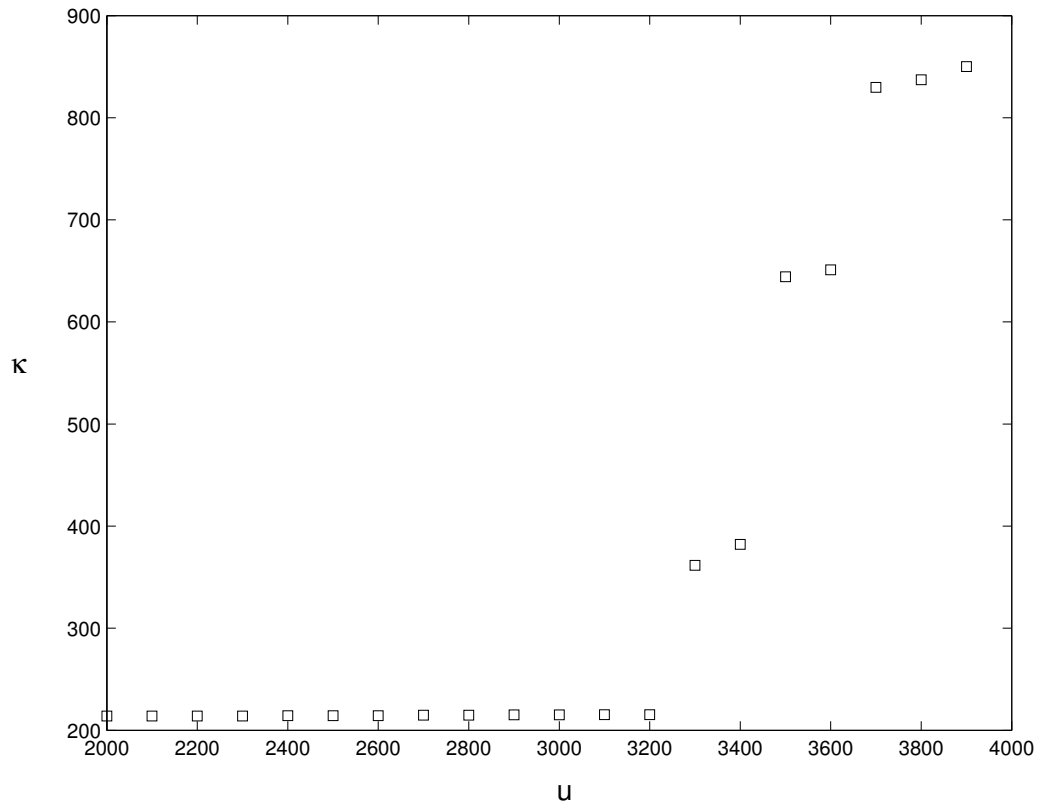


Figure 4.4: Condition number of Δ_{uu} (top) and $I - P_{uu}$ (bottom).

First, one can observe that the condition numbers of $I - P_{uu}$ are 10 times smaller than those of Δ_{uu} . This is to be expected since $I - P_{uu}$ is a “normalised” Δ_{uu} . As a result, conjugate gradients are faster in the case of $I - P_{uu}$, which is confirmed by Figure 4.3. However, performance is closer to linear with Δ_{uu} , because of the additional workload of the nonsymmetric algorithms. A second interesting fact is that the methods are not equally affected by fluctuations in κ . As seen in Figure 4.4, κ can be radically different for different values of u , since only changes in the extremal eigenvalues of the matrix have an effect on the value of κ . Figures 4.1-4.3 show that GMRES and BICGSTAB (which uses GMRES) are more sensitive to κ , whereas CG, MINRES and BICG appear to benefit from superlinear convergence.

The experiments also confirm that BICG and QMR are less efficient than CG and MINRES respectively with a symmetric coefficient matrix. It is a little surprising, though, that MINRES and SYMMLQ perform better than CG. A probable explanation is that Δ_{uu} is weakly positive definite, which makes gradient descent on $x^T Ax$ harder. As a whole, the performance of the methods applied appears to confirm the expected $O(u^\omega)$ ($1 < \omega < 2$) behaviour. CGS with $I - P_{uu}$ is the fastest while being reasonably stable. MINRES and SYMMLQ with Δ_{uu} are next best in terms of performance and are very stable.

4.3 Preconditioning

Most of the time conjugate gradient methods are combined with a process called preconditioning, which enhances the condition number of the linear system and is thus expected to lead to better convergence. For a system $Ax = b$, a preconditioner is a matrix M , which can be selected in different ways, according to the preconditioning method. Usually, M is chosen to be an approximation of A such that the matrix $M^{-1}A$ can be computed

fairly cheaply. Having computed $M^{-1}A$, the system $(M^{-1}A)x = (M^{-1}A)b$ can be solved using a conjugate gradient method. If M is appropriately chosen, $M^{-1}A$ will be better conditioned than A . This scheme can even apply to symmetric methods, with slight modifications in their algorithm, although $M^{-1}A$ may be nonsymmetric. Other preconditioning methods may involve two preconditioners M_1, M_2 or an *inverse preconditioner*, i.e. one that approximates A^{-1} . Here, only a few simple preconditioners of the standard type are examined.

The main concern when choosing a preconditioner is the tradeoff between one-off costs of computing $M, M^{-1}A$ and the savings from faster convergence of the new system, especially when multiple systems with the same coefficient matrix are to be solved. The simplest preconditioner, called Jacobi preconditioner, is simply a diagonal matrix consisting of the diagonal elements of A . It has low construction and inversion costs, but offers only a crude approximation of A . However, in the case of $A = I - P_{uu}$ and particularly when A is not too sparse, non-diagonal elements are generally small compared to the diagonal ones. Some investigative runs show a significant improvement with Jacobi preconditioning (Figure 4.5). Here, $M^{-1}A$ was computed once before the runs and only the cost of the iterative methods is shown.

Other more complicated methods are based on incomplete factorisations of matrices. Complete factorisations are procedures like *LU decomposition* or *Cholesky decomposition*. LU decomposition has the form

$$A = LU,$$

where L is a lower triangular and U an upper triangular matrix. Cholesky decomposition applies to symmetric positive definite matrices only and has the form

$$A = LL^T,$$

where L is lower triangular. Both are used for solving linear systems exactly,

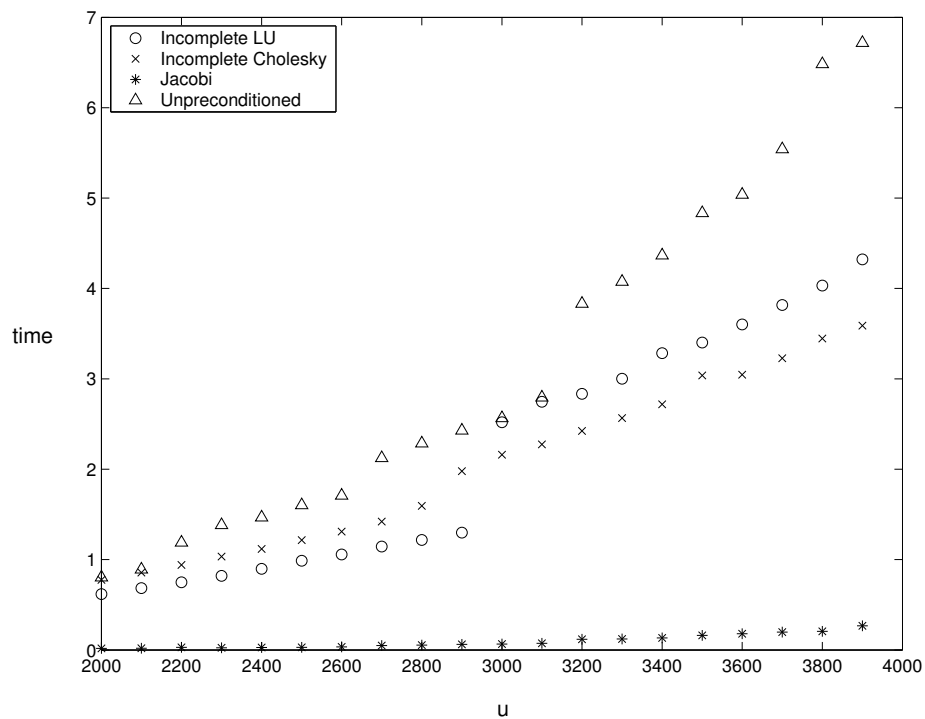


Figure 4.5: Required time for iterative methods after preconditioning. The systems are the same as in Figures 4.1-4.3 and CGS is also shown for comparison. Jacobi and incomplete LU preconditioning were combined with CGS on $I - P_{uu}$, while incomplete Cholesky preconditioning with MINRES was applied to Δ_{uu} .

Cholesky decomposition being the faster approach, but their time complexity is at least quadratic (see (Reif, 1998)). Their incomplete versions are used for approximate solutions of sparse linear systems and as preconditioners for iterative methods. They also yield lower and upper triangular matrices whose product, however, is not exactly equal to A . Furthermore, the approximation is less sparse than A , in other words there is *fill-in* in some of A 's nonzero positions. The degree of fill-in can be specified with a *drop tolerance* parameter, thus allowing only small enough values in the fill-in positions (see (Barrett et al., 1994)). The smaller the drop tolerance, the better the approximation and the more the construction cost. After a factorisation has been obtained, inversion and multiplication is fast because of the triangular form of the factors.

In Figure 4.5, results of incomplete LU and Cholesky preconditioning tests are shown. The drop tolerances were set to 10^{-5} after measuring the approximation error for different tolerances. For smaller values, the approximation improvements become more and more expensive to get. The effect of preconditioning is similar in both cases, but the improvement over unpreconditioned methods is about 10 times better under Jacobi preconditioning. In weight learning with gradient descent, the savings by Jacobi preconditioning were observed to be approximately 30% in total, there being a 40% improvement in conjugate gradient convergence.

4.4 Weight Learning

As a result of the previous analysis, the gradient descent for learning weights is modified so that CGS is applied to the system (2.4) after a Jacobi preconditioner is computed (once per iteration of the gradient descent). The sparsity pattern of W remains the same for all iterations. Although different values of σ_d should normally alter the sparsity structure, the computational

constraints force the approximating assumption that nearest neighbours do not change.

Chapter 5

Experimental Results

5.1 Weight Matrix Construction

The implementation presented in the previous chapters was tested with experiments performed on some digits data sets, including the ‘1-2’ and the ‘0-9’ data sets used in (Zhu et al., 2003a). All tests were done on an 1.8GHz 64bit Opteron with 2GB of RAM.

First, I have compared the time cost of constructing the sparse weight matrix using PCA and kd-trees with a simple Matlab algorithm that computes the weights for all i, j and then sparsifies the matrix. Both k NN and ϵ NN weightings were constructed and the results are shown in Figure 5.1.

As expected, the time increases linearly with k and quadratically with ϵ . Moreover, the savings in the latter case are more significant. With k NN, there are savings only if $k < 8$. The reason is that a larger k in kd-tree search keeps the radius of the search so large that most hyperrectangles are examined.

A similar experiment was run with a larger data set, obtained from the MNIST data set (MNIST handwritten digit database,). The sample consisted of 20000 points extracted from the MNIST training set of 60000 points. Due to memory limitations, only a 3NN and a 5NN matrix were constructed

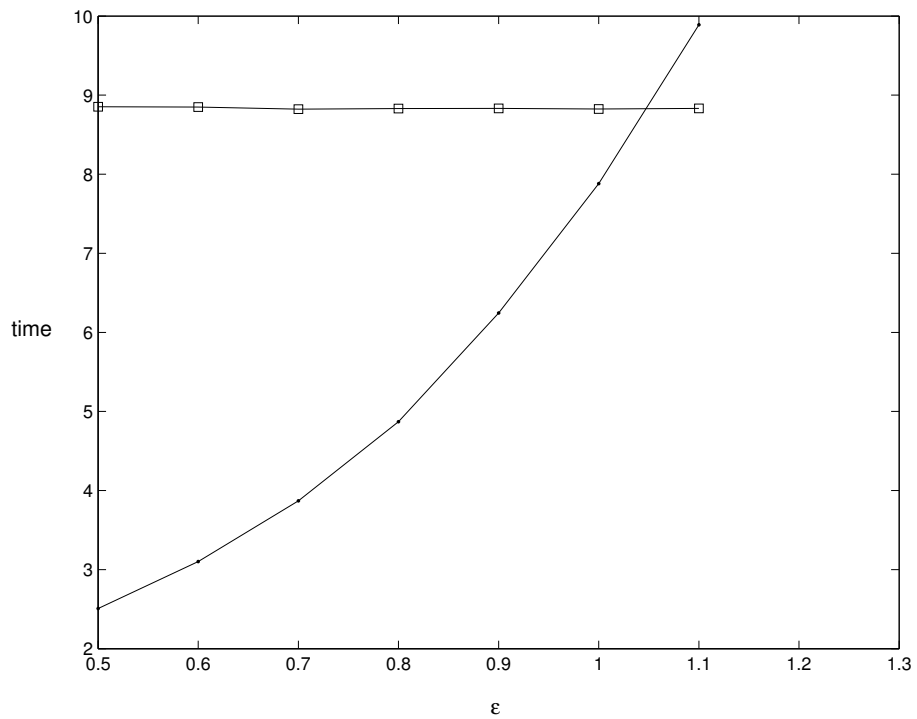
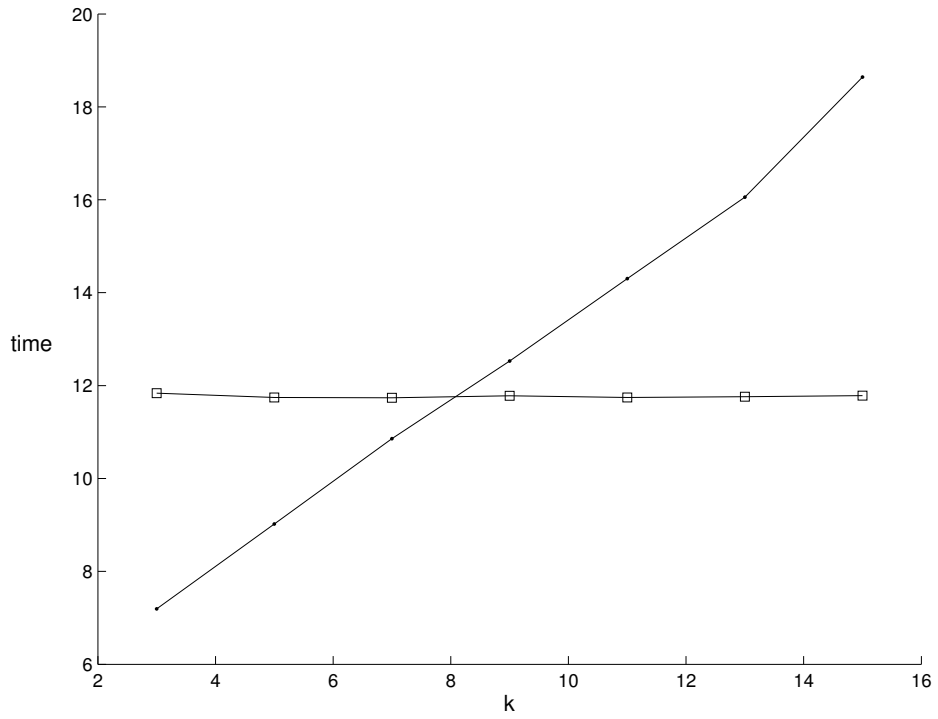


Figure 5.1: Cost of constructing a weight matrix from the '0-9' data set (4000 points). On top, the matrix was exp-weighted kNN and at the bottom exp-weighted ϵ NN. Squares correspond to a standard sequential algorithm and dots to the kd-tree algorithm.

and the times were about 120 and 290 seconds respectively.

5.2 Classification

Having constructed sparse weight matrices, the efficiency of computing labels for the unlabeled points can be tested. Figures 5.2 plot the error of the harmonic solution versus time. Both the ‘1-2’ and ‘0-9’ data sets were used and l ranged between 20 and 100. For each l , the plotted value is the average over 25 random reshuffles of the points. Length scales σ_d were equal to 380 in the first case and 400 in the second. As we see, first, the conjugate gradient runs have exactly the same accuracy of classification, which means that any inaccuracies of the solution were too negligible to change the labelling. Secondly, the speed of conjugate gradient is much better than that of the direct method apart from the cases with few labeled points, when the unlabeled Laplacian is close to singular and iterative methods converge more slowly.

With the MNIST-based set (Figures 5.3, 5.4), conjugate gradient based classification is better with kNN than with exp-weighted graphs, as expected. Here, as in Figure ??, we see that time and error are not inversely. This is due again to ill-conditioning with few labeled points.

5.3 Weights Learning

To assess the effect of iterative methods in the weight learning process, I started gradient descent from the same initial $\sigma_d = 380$ using preconditioned CGS first and then a direct solver. The ‘1-2’ set was used, with 100 labeled points, the smoothing parameter was $\epsilon = 0.1$ and the tolerance for CGS was 10^{-6} . The plot of entropy against time can be seen in Figure 5.5. With CGS search seems to move about 4 times faster, but is not as smooth as with the standard method. Smoothness can be adjusted through the tolerance

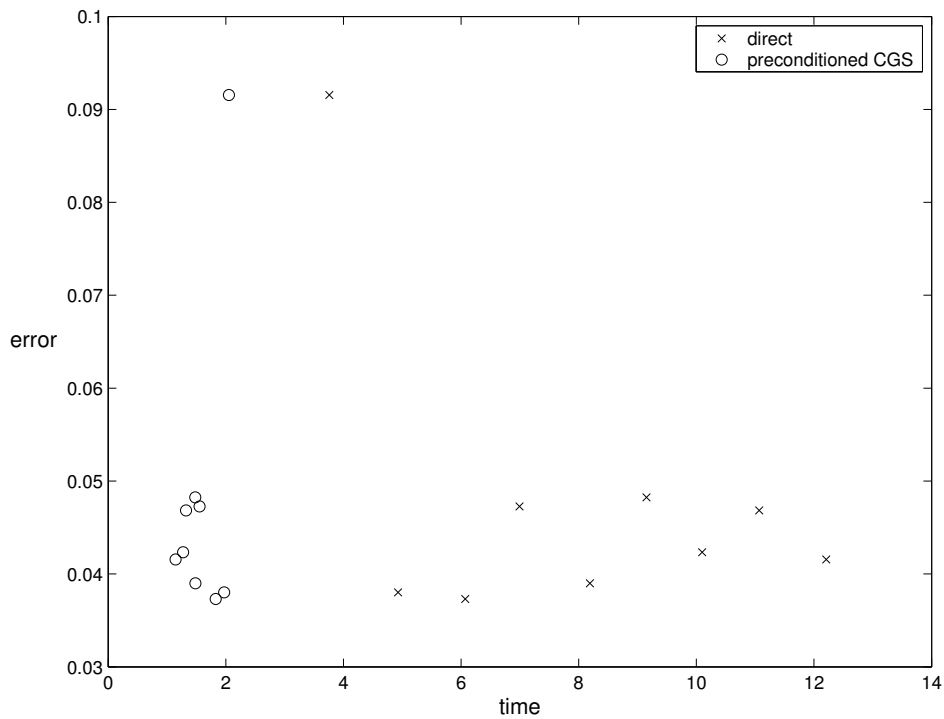
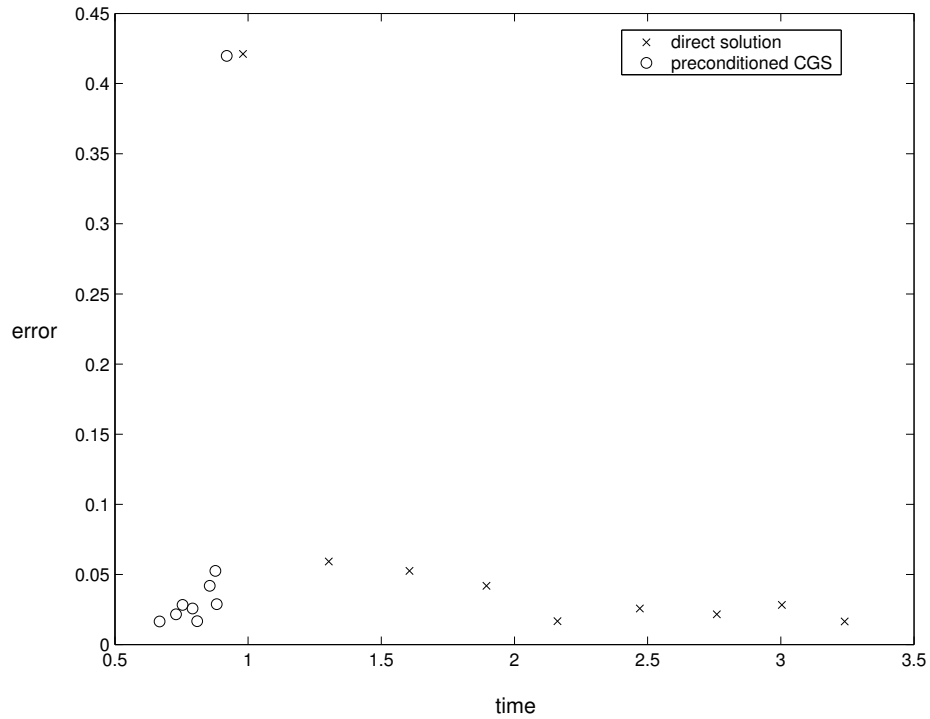


Figure 5.2: Classification error versus time cost for preconditioned CGS and the standard linear system solver. On top, the '1-2' data set is used and at the bottom, the '0-9' data set (classifying digit '5'). The graphs were exp-weighted kNN with $k = 10$ and the tolerance of CGS was 10^{-6} .

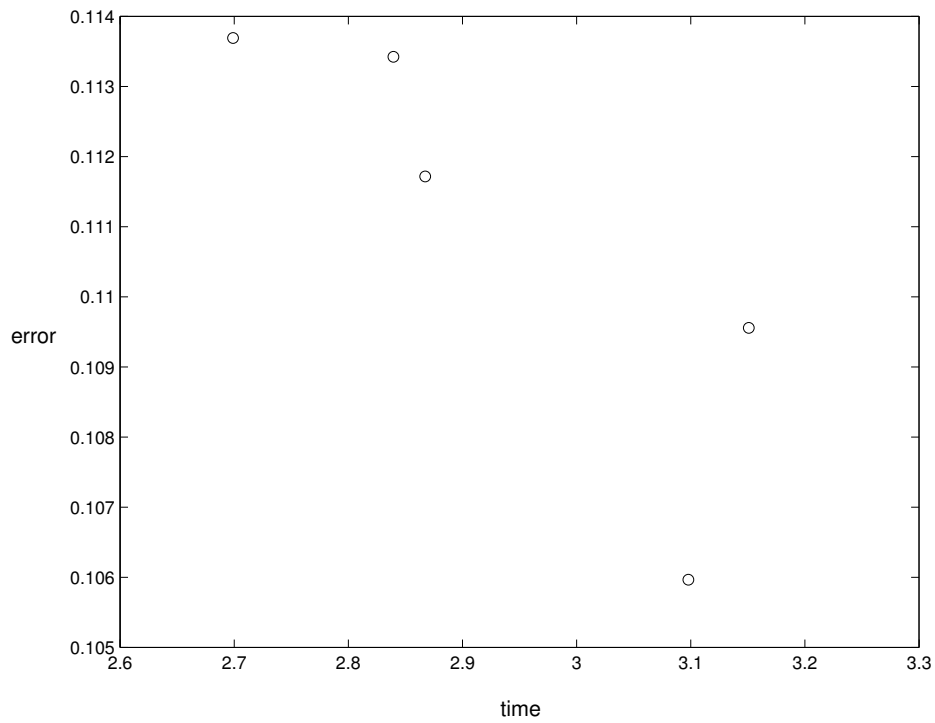
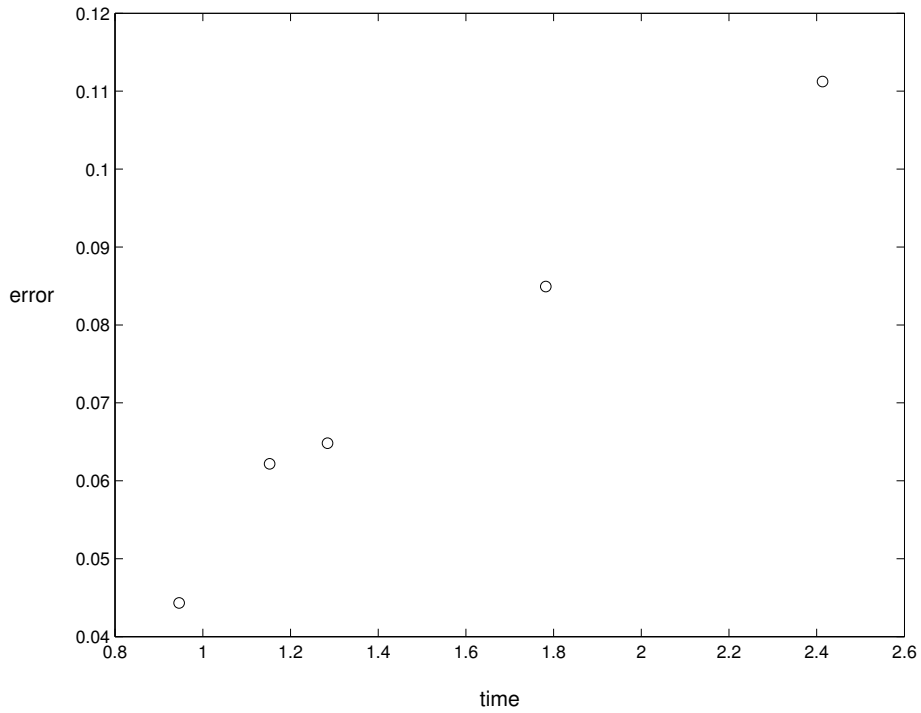


Figure 5.3: Classification error versus time cost for preconditioned CGS with 20000 points from the MNIST data set (classifying digit ‘3’). On top, unweighted kNN and at the bottom, exp-weighted kNN ($k = 3$). l ranges from 100 to 900. Length scales were equal to 400 and the tolerance of CGS was 10^{-6} .

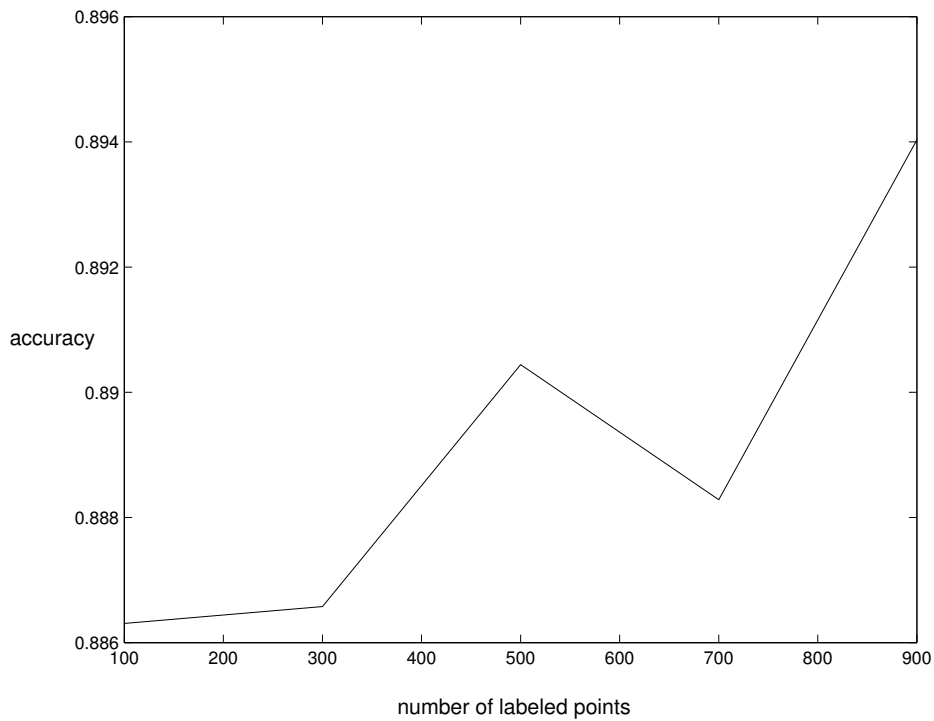
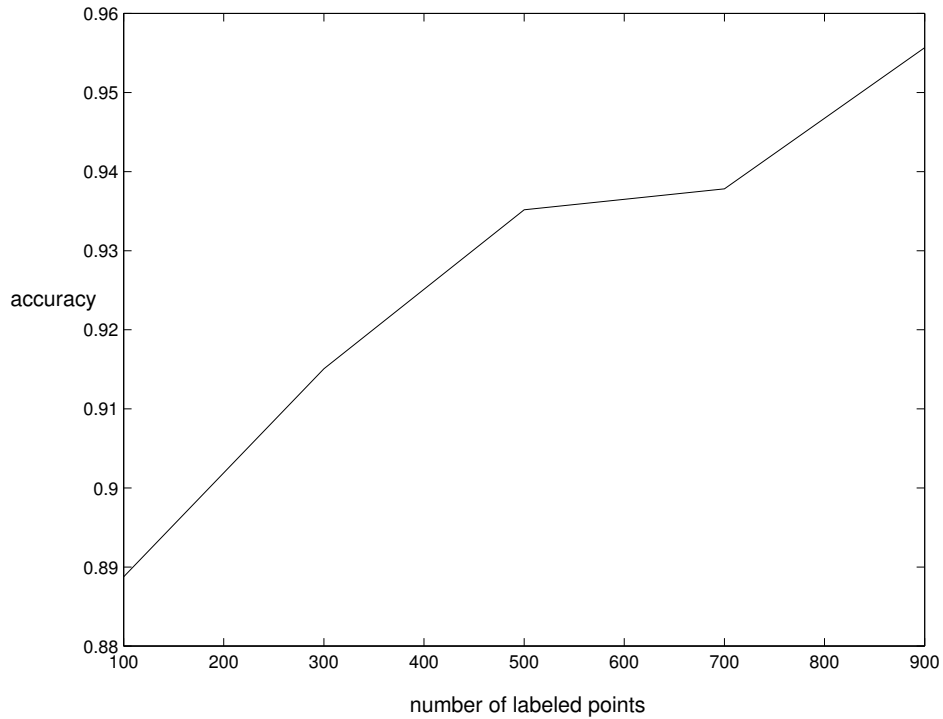


Figure 5.4: Accuracy versus l for preconditioned CGS with 20000 points from the MNIST data set (classifying digit ‘3’). On top, unweighted kNN and at the bottom, exp-weighted kNN ($k = 3$). Length scales were equal to 400 and the tolerance of CGS was 10^{-6} .

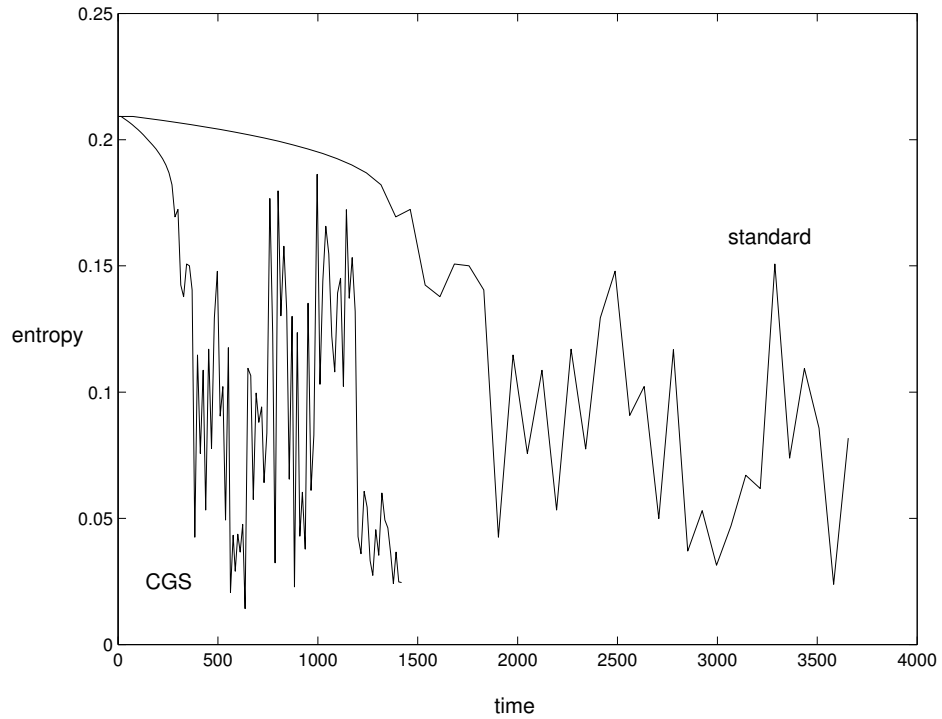


Figure 5.5: Entropy versus time for two weight learning runs, one with CGS and one with the standard direct solver.

parameter or the learning rate but it may not be as important if the entropy decreases on average.

Chapter 6

Conclusion

The implementation presented successfully constructs a sparse weight matrix for a graph and is observed to enhance the speed of classification and hyperparameter learning. The kd-tree approach for computing nearest neighbours and weights is fast only with small k , but enables computation with larger sample sizes. PCA preprocessing seems to combine smoothly with kd-trees and incurs no significant cost.

Of the conjugate gradient methods, CGS performs consistently well on the problem considered here. Other methods (MINRES and SYMMLQ) are also good alternatives. Preconditioning, even of a simple type, should always be used to improve performance of such methods. Conjugate gradients are faster than standard methods in most cases, without decrease in accuracy. However, there are not significant computational savings when the number of labeled points is very small.

For learning the weights, using iterative methods improves speed of convergence but adds more fluctuations. The speed of this process, however, remains slow when the dimensionality is high and other methods could be tried. Low-rank approximations may not be a good alternative, since a brief investigation has shown discouraging results, but other more complicated matrix approximations may work.

References

- Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and Van der Vorst, H. (1994). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA.
- Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. In *Communications of the ACM*, volume 18, pages 509–517.
- Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press.
- Concus, P., Golub, G. H., and O’Leary, D. P. (1976). A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations. Technical Report STAN-CS-76-533, Stanford University, Computer Science Department.
- Friedman, J. H., Bentley, J. L., and Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226.
- Hestenes, M. R. and Stiefel, E. (1952). Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436.

- Matlab Documentation. Simultaneous linear equations, sparse matrices. <http://www.mathworks.com/access/helpdesk/help/techdoc/math/sparse19.html>.
- MNIST handwritten digit database. <http://yann.lecun.com/exdb/mnist/>.
- Moore, A. (1990). *Efficient memory-based learning for robot control*. PhD thesis, University of Cambridge.
- Parlett, B. N. (1998). *The Symmetric Eigenvalue Problem*. Society for Industrial and Applied Mathematics.
- Reif, J. (1998). Efficient approximate solution of sparse linear systems. In *Computers and Mathematics with Applications*, volume 36, pages 37–58.
- Shewchuk, J. R. (1994). An introduction to the conjugate gradient method without the agonizing pain. School of Computer Science, Carnegie Mellon University.
- Smola, A. and Kondor, R. (2003). Kernels and regularization on graphs. In *Conference on Learning Theory, COLT/KW*.
- van der Sluis, A. and van der Vorst, H. A. (1986). The rate of convergence of conjugate gradients. In *Numerische Mathematik*, volume 48, pages 543–560. Springer-Verlag.
- Weber, R., Schek, H.-J., and Blott, S. (1998). A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24th VLDB Conference*.
- Zhu, X., Ghahramani, Z., and Lafferty, J. (2003a). Semi-supervised learning using Gaussian fields and harmonic functions. In *Proceedings of the 20th International Conference on Machine Learning*.

Zhu, X., Lafferty, J., and Ghahramani, Z. (2003b). Semi-supervised learning: from Gaussian fields to Gaussian processes. Technical Report CMU-CS-03-175, CMU.