Sparse Learning Package with Stability Selection

And Application to Alzheimer's Disease

by

Ramesh Thulasiram

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved June 2011 by the
Graduate Supervisory Committee:

Jieping Ye, Chair
Guoliang Xue
Arunabha Sen

ARIZONA STATE UNIVERSITY

August 2011

ABSTRACT

Sparse learning is a technique in machine learning for feature selection and dimensionality reduction, to find a sparse set of the most relevant features. In any machine learning problem, there is a considerable amount of irrelevant information, and separating relevant information from the irrelevant information has been a topic of focus. In supervised learning like regression, the data consists of many features and only a subset of the features may be responsible for the result. Also, the features might require special structural requirements, which introduces additional complexity for feature selection. The sparse learning package, provides a set of algorithms for learning a sparse set of the most relevant features for both regression and classification problems. Structural dependencies among features which introduce additional requirements are also provided as part of the package. The features may be grouped together, and there may exist hierarchies and overlapping groups among these, and there may be requirements for selecting the most relevant groups among them.

In spite of getting sparse solutions, the solutions are not guaranteed to be robust. For the selection to be robust, there are certain techniques which provide theoretical justification of why certain features are selected. The stability selection, is a method for feature selection which allows the use of existing sparse learning methods to select the stable set of features for a given training sample. This is done by assigning probabilities for the features: by sub-sampling the training data and using a specific sparse learning technique to learn the relevant features, and repeating this a large number of times, and counting the probability as the number of times a feature is selected. Cross-validation which is used to determine the best parameter value over a range of values, further allows to select the best parameter value. This is done by selecting the parameter value which gives the maximum accuracy score. With such a combination of algorithms, with good convergence

guarantees, stable feature selection properties and the inclusion of various structural dependencies among features, the sparse learning package will be a powerful tool for machine learning research. Modular structure, C implementation, ATLAS integration for fast linear algebraic subroutines, make it one of the best tool for a large sparse setting. The varied collection of algorithms, support for group sparsity, batch algorithms, are a few of the notable functionality of the SLEP package, and these features can be used in a variety of fields to infer relevant elements. The Alzheimer Disease(AD) is a neurodegenerative disease, which gradually leads to dementia. The SLEP package is used for feature selection for getting the most relevant biomarkers from the available AD dataset, and the results show that, indeed, only a subset of the features are required to gain valuable insights.

ACKNOWLEDGEMENTS

Thanks to my advisor Dr. Jieping Ye, for his patience, help, and incredible support throughout the term of my masters. Without Dr. Ye's guidance and help, this would not have been possible. Thanks also to my committee members and class teachers Arunabha Sen and Guoliang Xue for their patience and inspiration.

I would also like to use this opportunity to thank Jun Liu, Ji Liu, Jianhui Chen, Shuo Xiang, Liang Sun, Shuiwang Ji, Lei Yuan, Betul Ceran, Qian Sun. Also discussions with Ashok Venkatesan, Prasanna Sattigiri and Mohit Shah had really been useful. And thanks to my friends Kumaraguru Paramasivam, Abhishek Kohli, Pravin Dalale, Aditi Vadodkar, Jawahar Ravee Nithiyanandam and Anil Reddy.

I would like to thank everyone who supported me in anyway during my study at ASU. Last but not the least, I would like to thank my family for their immense support that they had given me throughout.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

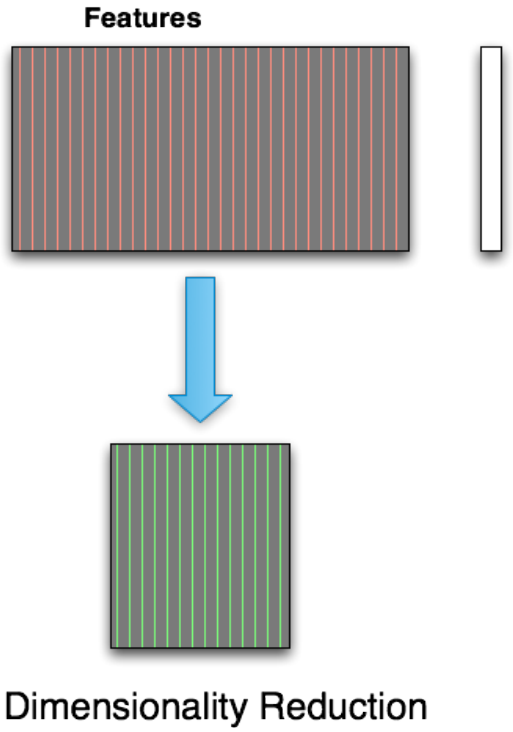
Machine learning is a branch of artificial intelligence which deals with learning from past data for future predictions. Being a relatively new field, with the increasing power of computers it is being used in almost every area dealing with empirical data. Machine learning is broadly classified as: Supervised Learning, Unsupervised learning, Semi-supervised learning, reinforcement learning. As these names suggest, they deal with uncertainties according to what information is available, and what is the maximum possible inference that can be done. The recent developments in computational speed and power, has led to the use of many of these techniques, which were previously not possible. It is worth mentioning here that there has been quite some interest in parallelizing these data and computation intensive algorithms, and many of these developments along with the increasing power of computers have led to such an interest. Learning from the past, is a useful way to reduce uncertainty of the future. Much of statistical learning, has focused on using existing data to predict uncertainty. Machine learning research has been an exciting field for more than 30 years in the past, and is still evolving, and will continue to be one of the primary research areas in the coming years, due to the numerous areas of application of machine learning techniques. Bio-informatics, predictive data analytics, economic analysis, web mining, buyer recommendation systems, social behavioural studies, and more and more areas have been finding machine learning to be highly practical science relevant to them. With the advancement of computing power, storage capacities, data flows, efficient algorithms for computations, efficient parallel and distributed computing; machine learning has found even more practical usage in every areas of science and technology.

Many of the real-world phenomena have a simple and comparatively com-

pact representation, in spite of being represented in a non-compact way. In practice, due to empirical uncertainty, we are overloaded with a lot of irrelevant information. The way to compactly represent and comprehend a phenomenon, has been the most fascinating aspect of science and mathematics. Apart from the prejudices of compact representation for human understanding and manoeuvrability, the representations of many real-world processes are indeed sparse and sparse representations has found useful applications in signal processing, neuroscience, bio-informatics, image processing, and many other fields.

Dimensionality reduction is the process of reducing the amount of data available, by cutting down the number of random variables under consideration, also known as the features. Based on how the features are selected, the algorithms are classified as feature extraction or feature selection; the former does it by giving probabilities to the features, and the latter by iteratively adding or removing features from a *select* or *remove* set thus selecting the optimal feature set.

Figure 1.1: Dimensionality reduction - reducing the number of features



Since we are uncertain about the models, there is a possibility of using the irrelevant information in our knowledge base, to make predictions. This turns out to be very costly, and much of the recent works have been focusing on reducing this unnecessary information.

Sparse learning is a relatively new area in machine learning, where new ideas are being evolved, and it is finding practical applications. Sparse learning focuses on using the available information to minimize the irrelevant information in the data. Selecting genes responsible for a particular disease, predicting user behaviour from only a subset of user data, finding regions in brain whose activities correspond to the causation of a particular disease are all some of the applications of sparse learning.

The SLEP package is aimed at providing many sparse learning routines, which are fast and easy to use. The main motivation for creating the software package, was to promote it as a tool for sparse learning applications, with its fast routines and graphical user interface. It also provides visualization of data. The stability selection is a model selection algorithm, which can be used for selecting the stable set of features. The path-wise solutions, and cross validation routines help the user choose among the different parameter values. We now give a brief overview of the algorithms that are part of the software.

The ℓ_1 norm regularization, has been the topic of focus for much of the research in sparse learning, due to their sparsity-inducing property, and strong theoretical guarantees. The ℓ_1/ℓ_q norm is another sparsity inducing norm, where the sparsity is focused towards group of features.

Here we discuss the following sparse learning algorithms,

- ℓ_1-norm regularized - considers both least squares loss function and logistic loss function with additional constraints such as non-negative solutions

- ℓ_1-norm constrained - similar to the regularized form, but the problem is formulated differently, the ℓ_1-norm is added as a constraint.
- Group Lasso - uses ℓ_1/ℓ_q-norm for group sparsity, available for both least squares loss and logistic loss
- Multi-task Lasso- uses ℓ_1/ℓ_q-norm for multi-task sparse solutions, available for both least squares loss and logistic loss
- Multi-class Lasso- uses ℓ_1/ℓ_q-norm for multi-class sparse solutions, available for both least squares loss and logistic loss, it is a general case of the group lasso
- Fused Lasso - here an additional penalty, which penalizes large successive difference in the solutions, hence the obtained solution is smooth as well as sparse - the penalty term used is called the fused penalty :
- Sparse group lasso - the sparse group lasso, adds an additional constraint which allows for sparsity within the groups, available for both least squares loss and logistic loss. Also there is an algorithm for multi-class sparse-group lasso, for use when there are multiple classes.
- Tree-structured group Lasso - here the groups further form hierarchies, and the features are tree-structured, this is also available for both least squares and logistic loss
- Overlapping group Lasso - group lasso, and sparse group lasso allow for group sparsity, but the groups cannot overlap. The tree-structured group lasso is a special case of overlapping group lasso. This is available for both least squares and logistic loss.

There are a few batch processing routines for feature selection and for setting optimal values for regularization constants.

- Stability Selection - selects the stable sets of features using one of the algorithms above, there are utility functions for extracting features after running this algorithms to get the probabilities of each of the features.

- Leave 1 Out cross validation - A cross-validation routine, which runs as many times as the number of data points (or rows), each time leaving one of the data point for test and using the remaining for training.

- k-Fold Cross Validation - A cross validation routine, which splits the data into k folds, runs k times, each time using a new fold for testing while the remaining is used for training.

- Bootstrapped Lasso - This is a routine which is used to find the best set of features for least squares loss, by running the least squares loss routines many times, with different bootstrapped samples each time.

The problems listed above are challenging to formulate, as the ℓ_1 norm, the ℓ_1/ℓ_q norm, the fused Lasso penalty, the sparse group Lasso penalty, the fused lasso penalty, the tree structured and overlapping group lasso penalties, are not smooth. As mentioned earlier, the problems are solved by formulating them as convex optimization problems which have a single global solution. As discussed in [34], it is almost impossible to find the exact solution to any of these problems. We use an iterative algorithm, as discussed in [34], which minimizes upto an error margin of $1/k^2$ for the number of iterations k. The sparse learning in general, has many applications, as it reduces the original representation by a considerable amount. More and more fields are considering the use of sparse learning, and also introducing additional complexities and theory. These have lead the the development of sparse models for least squares regression, principal component analysis, support vector machines, and logistic regression.

The ℓ_1/ℓ_q norm regularization, belonging to the composite absolute penalties family of regularizers, facilitates group sparsity which has applications in computer vision, graphics, fraud detection, and many such fields. The fused Lasso regularizer, emphasizes a smooth neighbourhood of values for the weights, and this can be used in image de-noising, prostate cancer analysis, genomic hybridization, and time-varying networks, where the ordering of features is possible.

The sparse group lasso, is an extension of group lasso and lasso, which achieves the within and between group sparsity simultaneously, meaning, many groups of features are zero, and also those non-zero groups of features can be made sparse. It is a special case of the tree group lasso. Again, many can see the applicability of this technique to their particular area. Tree Structured group lasso, is another powerful sparse learning routine, but here, the features are grouped, and also have hierarchies. The leaf nodes are features, and internal nodes are clusters of features, the structural regularization is based on the group lasso penalty, which will be discussed in detail in later sections. This can be used mainly in image processing, and multi-task learning. Some applications desire an overlapping group structure in the features, which has not yet been addressed by the above algorithms. For this, we provide the overlapping group lasso, which can solve problems which organizes the features in groups, and which are overlapping.

Stability selection [31], is a work on feature extraction, which focuses on collecting the best set of features, given the data and its dependent variables. This has been proven to have strong theoretical properties, and empirically the results have been good as well. It works by using an existing sparsity inducing routine, to get solutions for a sub-sample of the features. The *stability path* for the features is drawn, which is a path for a particular feature for the frequencies of this parameter being non-zero over all the runs with different random sub-sample (drawn with replacement), drawn through different parameter settings for the sparsity inducing

parameter. Once we compute this path, we get the maximum probability of each feature over its stability path, and use this as a measure of how probable this feature is likely to get selected. This makes much sense intuitively, but has been proved to be theoretically good as well. Using stability selection, can drastically improve the feature extraction process, and can reduce doubts about the effectiveness of sparsity inducing parameters.

The bootstrapped lasso, is an method to select the best set of features for the regression problem. This makes use of the lasso, with random bootstrapping samples, run many times, and selects those features which are constant over many runs of the different bootstrap samples.

This package in C, provides all these algorithms with a good design, scalable, and natively running implementation compared to the Matlab package which contains many of the algorithms discussed above. And, these have additional support for extracting features, running batch algorithms, quick visualizations, and hence will be a very useful companion in a scientific setting. The current implementation of these algorithms has faster running time than in Matlab, with exact precision as Matlab, and many times very fast running times. This is especially useful for the batch algorithms as the Stability Selection, Bootstrapped Lasso and Cross Validation.

Chapter 2

RELATED AND REFERENCED WORKS

This chapter, is intended to give a brief review of all the work on sparse learning, and everything related to this paper. The reader is warned that, the content assumes the reader is well-versed with the sparse learning terminologies. The reader may come back after reading the rest of the document before starting with this.

For a general overview about convex optimization, [6] gives an excellent introductory explanation, with some convex analysis, formulating problems as convex optimization problems, then goes discussing about applications and many algorithms for convex optimization problems. Also, [34] is another set of useful material from lectures on convex optimization, and it gives even more insights into solving convex optimization problems efficiently. [32] discusses some of the efficient ways to solve convex programming problems, and most of the work in this Software has come from an efficient way to solve convex programming problems discussed by Nesterov in [33]. Many of these problems are also non-smooth which warrants the use of non-conventional methods for optimization. The non-smooth problems are hard to solve, and [5] introduces some theory on nonsmooth optimization.

The idea of sparse learning, was first a spark from the paper by Tibshirani [38] on ℓ_1 norm and its sparsity inducing property. [38, 17] have given a good account of ℓ_1 norm for least squares regression. [22, 20] discuss about logistic regression and the ℓ_1 norm for sparsity. Shalev-Shwartz and Srebro give a nice account of the guarantees of sparsifiability in [36]. They show this by introducing a randomized procedure comparing it to the ℓ_1 norm induced sparsity and giving a bound on the amount sparsity.

All these methods require a way to efficiently project points onto convex ob-

jects in high dimensions. [26, 10] focus on efficient projections required for the sparse learning algorithms, solvable by formulating as a convex optimization problem. These in turn require euclidean root finding, which has been explored very well, and [35] can be used for an introductory text on this, with [26] giving a detailed account of this, and a linear time algorithm with respect to the sparse learning setting.

For group sparsities, [27, 30, 42, 43, 44, 15, 13] are some of the references which discuss about group sparsities. The composite absolute penalties, which are mainly used in group and multi-task settings are discussed in [43, 44]. The multi-task problem in [23, 8, 2] discuss about the convex formulations, and how to solve these efficiently. The sparse group lasso, hierarchical group lasso, and the employment of Moreau-Yosida regularization, are discussed in detail in [21, 29]. Also the hierarchical model selection in [43, 44, 15, 13, 12] give an account of hierarchical and overlapping groups.

[19, 9, 4, 41] discuss the linear algebraic routines, and how to implement them in the most efficient manner, with some discussion of data structures as well. Particularly, [41] discusses the idea of Automated optimizations targeted at particular architecture of the computer, and fast linear algebraic subroutine running times.[28, 26, 39, 40] talk about fused lasso, efficient ways to solve the problem.

The applications of sparse learning, are discussed in [40, 1, 7, 14]. These are varied fields from bio-informatics to signal processing. A look at various research databases, suggest that sparse learning has numerous applications in a variety of fields. Stability selection is a recent paper, which focuses on selecting the most stable set of features among others[16, 31]. Cross validation allows us to select a good parameter value, which will be critical for a good prediction[18, 37, 11]. The bootstrapped lasso is another method which allows for feature selection by using a novel approach[3].

Chapter 3

FORMULATIONS AND ALGORITHMS

This section introduces the various formulations of sparse learning methods, and the algorithm used to solve these problems. The ℓ_1 norm and its sparsity inducing property is discussed first, continuing with ℓ_1/ℓ_q norm and its applications to sparse learning. This is followed by the *lasso*, *sparse logistic regression*, *group lasso*, *fused lasso*, *multi task learning*, *multi-class learning*, *tree structured group learning* and *overlapping group learning*.

We use some of the techniques defined in [34, 32, 33, 6, 5] to solve the problems, by formulating each as an optimization method, with additional constraints. Additionally, the problems discussed below, belong to the convex optimization class, which requires the objective function to be convex (though not necessarily smooth), and the constraints to form a convex set. Convexity has been proved to be a useful property, and more and more problems are being formulated as convex optimization problems and being solved [6]. Many problems are not themselves convex, but have a convex approximation which would work well. The convex problems are easy to solve, as they have a unique minimizer, and hence the solutions of the convex optimization problems converge to that particular solution. The zero norm (or ℓ_0-norm) is the count of number of non-zero elements of a vector. But the ℓ_0 norm does not form a convex cone (geometrically this can be seen when drawn in the $d+1$ dimension). The closest convex norm that matches this functionality of selecting non-zero elements from zero elements, is theℓ_1 norm. And in fact, it does well to induce sparsity into the solution.

Furthermore, we explore the ℓ_q/ℓ_1 norm, discussed in [43, 44], and these are of interest in problems involving group sparsities. We discuss many problems of this genre, and the application of ℓ_q/ℓ_1 norm to it. The next section introduces

the ℓ_1 norm and its sparsity inducing property.

3.1 Sparsity and the ℓ_1 norm

It has been well established that the ℓ_1 norm induces sparsity while expressed along with the loss function or used as a constraint in the minimization problems. Most of the algorithms in this package uses ℓ_1 norm to find a sparse solution. The effect of ℓ_1 norm and ℓ_2 norm can be combined to form regularized and sparse solutions. Typically, the problems are of the form

$$\text{minimize } f(\boldsymbol{x}) + \lambda \|\boldsymbol{x}\|_1$$

or

$$\text{minimize } f(\boldsymbol{x}) \text{ s.t. } \|\boldsymbol{x}\|_1 < z$$

where $f(x)$ is the loss function and $\|.\|_1$ is the ℓ_1 norm, and λ is the parameter which controls the amount of sparsity, and z is the constraint. The following diagrams illustrate an intuitive explanation of why ℓ_1 norm induces sparsity into the solutions. The ℓ_2 norm constraint, intersects the level curves at a point where both the parameter values are non-zeros.

The ℓ_1 norm constraint, intersects the level curves of the objective function at a point where one of the parameter values vanishes(is zero). This effect is more in high dimensional feature space, and hence the sparsity inducing property of ℓ_1 norm. More information about this can be found in [38].

Figure 3.1: The ℓ_2 norm

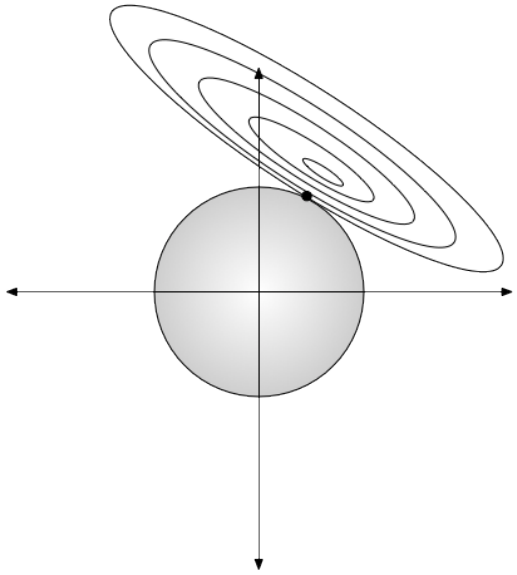
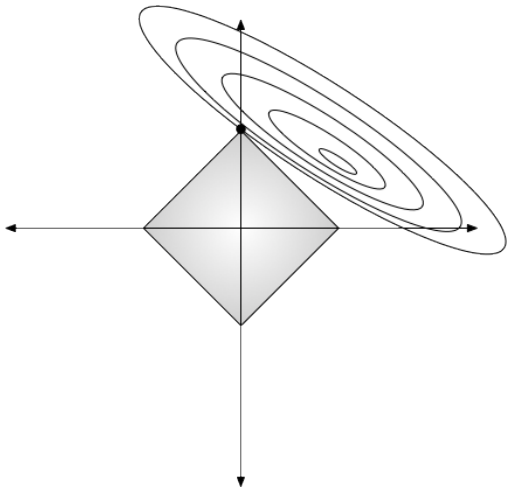


Figure 3.2: The ℓ_1 norm



3.2 The ℓ_1/ℓ_q-norm

For many of the problems involving a structure in their feature space, it is more convenient to introduce another norm, which allows the solutions to be structurally sparse. This follows from the ℓ_1 norms sparsity inducing property and is extended from the same. This has numerous applications specifically when there are relationship between features. This norm is represented as follows

$$(\sum_{g \in G} \|\boldsymbol{w}_g\|_2^q)^{1/q}$$

where each g represents a set of indices belonging to that particular group, and G is the set containing all such groups. Usually the value of q is ∞ or 2. For the ∞-norm we have the following;

$$\|\boldsymbol{w}\|_{1,\infty} = \sum_{g \in G} \max_k |w_{g,k}|$$

3.3 Efficient Projections

One of the key motivation to develop the sparse learning package is to use efficient projections of solutions on the constrained ℓ_1 norm polyhedra, which makes the algorithm, much more efficient. One of the advantages of the SLEP package is that it uses efficient projection algorithms, which take $O(n)$ time rather than the usual $O(n \log n)$ time for finding the projections onto the convex polyhedra. The SLEP package re-uses code developed in [24] for these routines. Many of them are built in C and are fast. The projections contain algorithms from the following [20, 25, 26, 27, 28, 23, 22]. Many of these techniques deal with sub-differentials, which are the differentials at a point which has more than one differential. Also duality is heavily made use of for finding the efficient projections.

Improved Bisection

The problem of Euclidean projections onto the ℓ_1 ball $B = \{\boldsymbol{x} : \|\boldsymbol{x}\|_1 \leq z\}$ can be represented as

$$\underset{\boldsymbol{x}:\|\boldsymbol{x}\|_1 \leq z}{\arg\min} \frac{1}{2}\|\boldsymbol{x} - \boldsymbol{v}\|^2$$

The Euclidean projection problem has been widely discussed and [25, 10] has proposed efficient ways to solve it by converting this problem as a zero finding problem. The problem of finding the euclidean projection of a vector of length n onto a closed convex set, especially the ℓ_1 ball and a few other polyhedra, play a vital role in solving many of the sparse learning problems. The improved bisection with warm restart shows much better empirical speed up than the competing ones. The worst case complexity of the improved bisection algorithm is linear, and hence this improves the overall performance of the sparse learning algorithms which depend on these projection problems. The algorithm first introduces a bisection algorithm, and improves on it by using the piecewise linear and convex structure of the polyhedra onto which the projection is made. The ℓ_1 ball constrained sparse learning algorithms especially use this algorithm to improve their efficiency.

ℓ_1/ℓ_q-Regularized Projection

The ℓ_1/ℓ_q-regularized Euclidean projection problem is an important step towards solving the ℓ_1/ℓ_q-norm regularized problem. The problem is efficiently solved by formulating the problem as 2 zero-finding problems as discussed in [27].

$$\underset{\boldsymbol{x}:\|\boldsymbol{x}\|_1 \leq z}{\arg\min} \frac{1}{2}\|\boldsymbol{x} - \boldsymbol{v}\|^2 + \lambda \sum_{i=1}^{s} \|x_i\|_q$$

The ℓ_1/ℓ_2 and ℓ_1/ℓ_∞ projection problems have been widely discussed and efficient algorithms have been proposed for them. But the extension to the general

case has been done by [27]. Again, bisection is used to solve the zero finding problem, and the special structure is made use of to find efficient solutions. Efficiently solving this projection problem is an important step in solving many of the sparse learning algorithms that will be discussed.

3.4 Non-negative Solutions

For classification and regression problems, an additional useful constraint which is often necessary for many problems is the non-negative constraint on the solution. This is particularly useful This is introduced as an additional constraint as,

$$\boldsymbol{x} \succeq 0$$

This limits the solutions to the first quadrant which is still convex and hence the problems domain is convex and needs little changes to the original algorithm. The non-negative solutions has found practical applications in sparse coding and a few other fields.

3.5 The Lasso

The lasso is one of the fundamental algorithms in the field of sparse learning. As mentioned above, it can be expressed as a constrained and regularized version, and the loss function is given by

$$||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}||_2^2$$

The SLEP package provides a regularization term for the ℓ_2 norm as well, which is controlled by a parameter λ_{ℓ_2}. The lasso is one of the seminal ideas on sparse learning, and has been the foundation for much of the sparse learning ideas till date. The lasso does not create subsets of features, but it tries to shrink variables

such that they are exactly zero. The SLEP implementation of the Lasso is very efficient and it uses the bisection algorithm for projections.

3.6 Sparse Logistic Regression

The logistic sigmoid function defined by

$$p(b|\boldsymbol{a}) = \frac{1}{1 + e^{-b(\boldsymbol{w}^T\boldsymbol{a}+c)}}$$

is a probability distribution function which can be used as a model for classification problems in machine learning. For given data $\boldsymbol{A} \in \mathbb{R}^{n\times p}$ and $\boldsymbol{b} \in \mathbb{R}^n$ and individual elements of $\boldsymbol{b}$ are in $\{+1, -1\}$, the negative log-likelihood function is given by

$$f(\boldsymbol{w}, c) = -\sum_{i=1}^{n} \log \frac{1}{1 + e^{-b_i(\boldsymbol{w}^T\boldsymbol{a_i}+c)}} \tag{1}$$

where, $\boldsymbol{w}$ and c are the parameters, which need to be minimized. The logistic model has the smallest probability of misclassification, given the data is from a logistic model with the given parameters.

3.7 Structured Sparsity

The feature space may sometimes need special requirements that allow the features to exhibit structural patterns. For example, in medical image data, different neighbourhoods may represent an object, and need to be represented in groups, either overlapping or not. And we can use the sparse learning to select a few groups and unselect other groups entirely to zero. Depending on the requirement one can use a particular structured sparsity. In the following subsections, we discuss the various structured sparsity algorithms which are available as part of the SLEP package.

Fused Lasso

The fused lasso is a particularly useful tool in applications which requires weights for adjacent features to be similar. This is accomplished by introducing a penalty which penalizes large deviation in weights for neighbouring features,

$$\sum_{i=1}^{d-1} |x_{i+1} - x_i|$$

and hence the fused lasso can be formulated as follows,

$$\text{minimize} \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_2^2 + \lambda_1 \|\boldsymbol{x}\|_1 + \lambda_2 \sum_{i=1}^{d-1} |x_{i+1} - x_i|$$

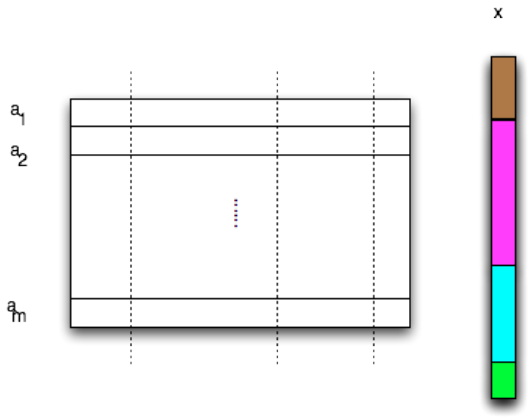
The value of λ_2 controls the deviation in neighbouring weights, and this combined with the ℓ_1 norm regularization gives a sparse solution. The figure below shows the solution obtained when using the `fusedLeastR` algorithm which will be discussed in the implementation section. Blue represents positive value, and magenta represents negative values. The amount of saturation of the color is varied, and it has full saturation for the largest positive or negative value, and very less saturation for values near zero. We can see that neighbouring values are fused or have values close to each other. The white boxes represent zeros.

Figure 3.3: Fused Lasso Solution

Group Lasso

Some problems in machine learning are such that the features form a group. So, the features can be arranged according to the group and each group can be of different sizes. Selecting groups by assigning weights to them, such that some of

Figure 3.4: Group Lasso



the groups have all their weights as zeros. This can be represented by the following,

$$\text{minimize} \frac{1}{2}\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}\|_2^2 + \lambda \sum_{i=1}^{g} d_i \|x_{G_i}\|_2$$

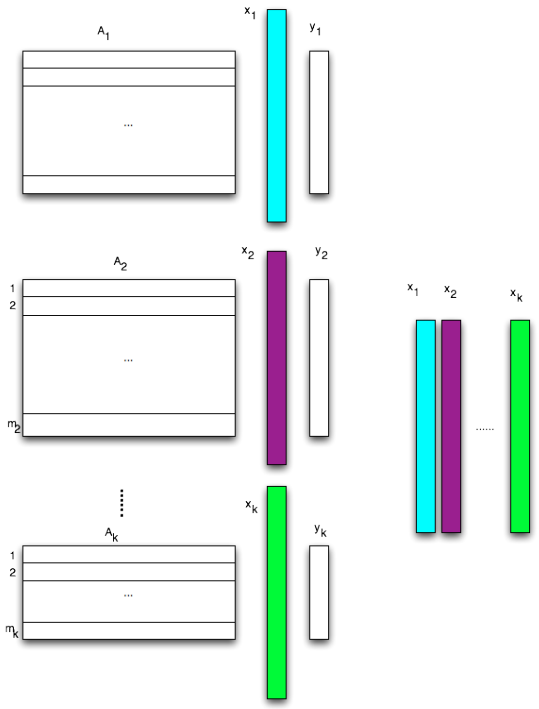
The logistic loss is used when the problem is used in a classification setting. The figure above illustrates the data matrix and solution partitioned as different groups.

Multi-task Learning

Sometimes we are presented with data which has multiple tasks combined together, and we are required to find the weight vector for each of the tasks. The following diagram illustrates the data matrix, observed results, and solution for multi-task learning. The results are partitioned according to the group information, and so are the rows of the data matrix corresponding to the result groups. And the solution vector is learned combined, and hence it is called multi-task learning. The multi-task least squares problem is given by

$$\text{minimize } \frac{1}{2}\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}\|_2^2 + \lambda \|x\|_{\ell_1/\ell_q}$$

Figure 3.5: Multi-task Learning



For the logistic loss the problem is similar but has the weights as another input. It is given by,

$$\text{minimize} \sum_{l=1}^{k} \sum_{i=1}^{m_l} w_{l_i} \log\left(1 + e^{-y_{l_i}(x_l^T a_{l_i} + c_l)}\right) + \lambda \|x\|_{\ell_1/\ell_q}$$

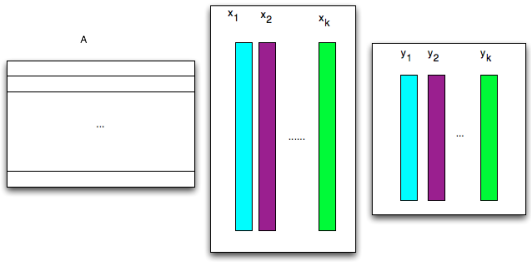
Multi-class Learning

The multi-class task learning is another important problem in machine learning where we are given results for different classes, and our task is to learn them together. This added with the ℓ_1/ℓ_q regularization term, results in a sparse solution the the multi-class learning problem. The figure above illustrates the multi-class learning problem.

The least squares multi-class problem is given by

$$\text{minimize } \frac{1}{2}\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}\|_2^2 + \lambda \|x\|_{\ell_1/\ell_2}$$

Figure 3.6: Multi-class Learning



and the multi-class logistic classification problem is given by,

$$\text{minimize } \sum_{l=1}^{k}\sum_{i=1}^{m} w_{il}\log\left(1+e^{-y_{il}(x_l^T a_{il}+c_l)}\right)+\lambda\|x\|_{\ell_1/\ell_q}$$

Sparse Group Lasso

There are cases when the solution needs to be group wise sparse, as well a need for sparsity of features even in groups which are selected. The sparse group lasso, introduces the additional ℓ_1 norm to the objective function to obtain sparse group solutions. The sparse group lasso problem can be formulated as

$$\text{minimize } \|\boldsymbol{x}-\boldsymbol{v}\|_2^2+\lambda_1\|\boldsymbol{x}\|_1+\lambda_2\sum_{i=1}^{g} w_i^g\|\boldsymbol{x}_{G_i}\|_2$$

For least squares and logistic loss this is given by

$$\text{minimize } \|\boldsymbol{A}\boldsymbol{x}-\boldsymbol{y}\|_2^2+\lambda_1\|\boldsymbol{x}\|_1+\lambda_2\sum_{i=1}^{g} w_i^g\|\boldsymbol{x}_{G_i}\|_2$$

$$\text{minimize } \sum_{i=1}^{m} w_i\log\left(1+e^{-y_i(\boldsymbol{x}^2\boldsymbol{a}_i+c)}\right)+\lambda_1\|\boldsymbol{x}\|_1+\lambda_2\sum_{i=1}^{g} w_i^g\|\boldsymbol{x}_{G_i}\|_2$$

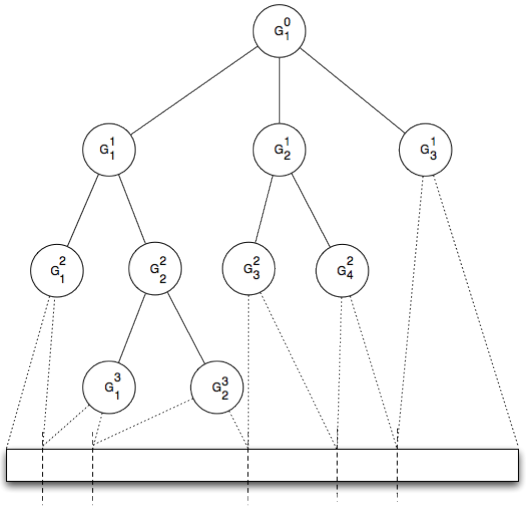
Also, the multi-class sparse group lasso with least squares loss is given by,

$$\text{minimize } \frac{1}{2}\|\boldsymbol{A}\boldsymbol{x}-\boldsymbol{y}\|_2^2+\lambda_1\|\boldsymbol{x}\|_1+\lambda_2\|x\|_{\ell_1/\ell_q}$$

Tree Structured Group Lasso

The tree structured group lasso, as the name suggests, has structures which are hierarchical, and the bottom most groups as the leaf groups are the smallest groups, and contain partitions of the features dimensions. The higher level groups, contain other leaf groups, and it goes all the way to the top, where the root node contains the next level nodes which complete the tree which covers all the feature dimension. The following diagram illustrates the tree structure of the groups. In facial recognition, one might need to group features belonging to different components of the face hierarchically, which will result in a tree structure among the features.

Figure 3.7: Tree Structure of the features



The regularization problem associated with the tree structured group lasso can be formulated as the following problem.

$$\text{minimize } \frac{1}{2}\|\boldsymbol{x}-\boldsymbol{v}\|_2^2+\lambda\sum_{i=0}^{d}\sum_{j=1}^{n_i}w_j^i\|\boldsymbol{x}_{G_j^i}\|$$

The least squares loss and logistic loss problems for tree structured group learning are represented as follows:

$$\text{minimize } \frac{1}{2}\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}\|_2^2 + \lambda \sum_{i=0}^{d} \sum_{j=1}^{n_i} w_j^i \|\boldsymbol{x}_{G_j^i}\|$$

$$\text{minimize } \sum_{i=1}^{m} w_i \log\left(1 + e^{-y_i(\boldsymbol{x}^2 \boldsymbol{a}_i + c)}\right) + \lambda \sum_{i=0}^{d} \sum_{j=1}^{n_i} w_j^i \|\boldsymbol{x}_{G_j^i}\|$$

The tree structured multi-task learning represented as above, forms a tree among the k tasks and solves the problem as above. Similarly we also have the multi-task logistic loss, multi-class least squares loss, and multi-class logistic loss for the tree structured problem, with the same regularization term and the tree is formed among the k tasks to be solved.

Overlapping Group Structured Lasso

The overlapping group lasso, allows for overlapping groups which introduce additional challenges for solving, which are discussed in [44]. In cases where we need group sparsity, which are also overlapping, we go for the overlapping group structured lasso. The tree lasso is a special case of the overlapping group lasso. The regularization problem associated with the overlapping group lasso problem is described as follows.

$$\text{minimize } \frac{1}{2}\|\boldsymbol{x} - \boldsymbol{v}\|_2^2 + \lambda_1 \|\boldsymbol{x}\|_1 + \lambda_2 \sum_{i=1}^{g} w_i^g \|\boldsymbol{x}_{G_i}\|_2$$

The least squares loss and logistic loss formulations for the overlapping group learning problem is given as follows.

$$\text{minimize } \frac{1}{2}\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}\|_2^2 + \lambda_1 \|\boldsymbol{x}\|_1 + \lambda_2 \sum_{i=1}^{g} w_i^g \|\boldsymbol{x}_{G_i}\|_2$$

$$\text{minimize } \sum_{i=1}^{m} w_i \log\left(1 + e^{-y_i(\boldsymbol{x}^2 \boldsymbol{a}_i + c)}\right) + \lambda \sum_{i=1}^{g} w_i^g \|\boldsymbol{x}_{G_i}\|_2$$

3.8 Cross Validation

The cross validation is an useful method for finding the best parameter values for a given sparse learning method, for a given training data. As previously mentioned, the data is divided into k folds, and each time a fold is picked for testing while the others are used to train the model. Then this test set is tested against the model trained using the rest of the data. Usually this is done over the parameter space by setting the parameter to different values and repeating the process. The accuracy / root mean square error is calculated for each value of the parameter. The one which gives highest accuracy or the least error would be a good choice for the parameter. If there are many parameters for a particular problem, then the number of times that we need to run the algorithm increases exponentially on the number of parameters.

3.9 Stability Selection

Stability selection addresses the problem of proper regularisation with a very generic sub-sampling approach [31]. Bootstrapping would also work. Beyond the issue of choosing the amount of regularisation, the sub-sampling approach yields a new structure estimation or variable selection scheme. For the more specialised case of high-dimensional linear models, the sub-sampling in conjunction with ℓ_1-penalised estimation requires much weaker assumptions on the design matrix for asymptotically consistent variable selection than what is needed for the (non-sub-sampled) ℓ_1-penalty scheme.

Chapter 4

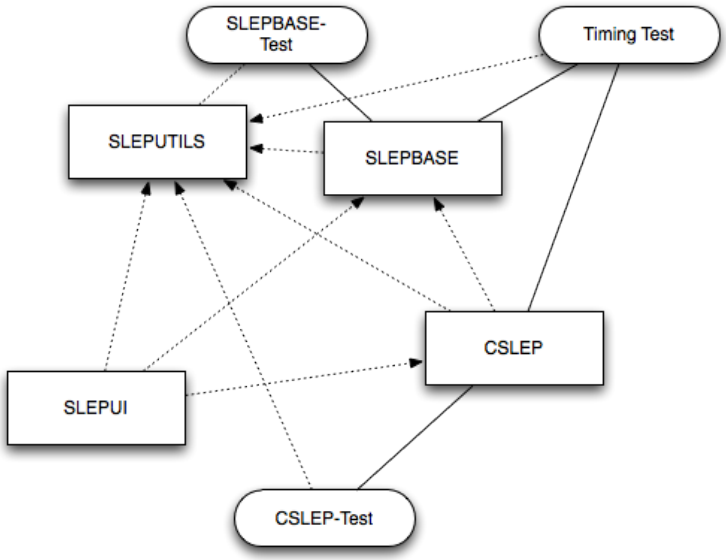
IMPLEMENTATION

This section starts with understanding the organization of the package, then explains the basic data structures used, the numerical routines which make it one of the fastest implementation available, and the visualization and user interface available.

4.1 Organization of the package

The original motivation for the package was to develop a C infrastructure for sparse learning, which will provide a strong and easy framework for the current implementation and future additions of sparse learning algorithms involving many of the dense and sparse linear algebraic formulations. The package is structured into different modules which incrementally and independently provide a base for the other modules. In this section, we will understand the organization of the package, the functionality of each module, and future functional extensions.

Figure 4.1: Organization of the SLEP package

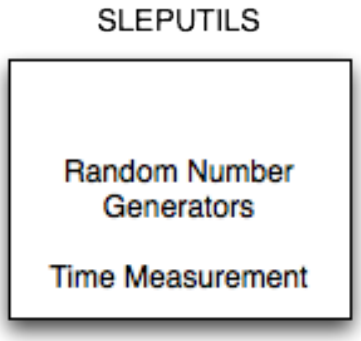


The diagram above shows the basic organization of the package, the details are discussed in the following subsections. The arrows indicate dependencies, and test routines are drawn in rounded rectangles.

Utilities and Base

The utilities module holds the random number generating and time measurement routines. These are key to some of the algorithms like Stability selection, bootstrapped lasso and also for testing as they allow us to measure time and generate data for testing. There is a random permutation routine which builds random permutation of integer arrays, which are used in sub-sampling, which is being used in stability selection. The time measurement is provided at the nanoscale, and is specifically tuned for Windows, Mac OS X and Linux. However, the accuracy of the time is dependent on the implementation at the OS level.

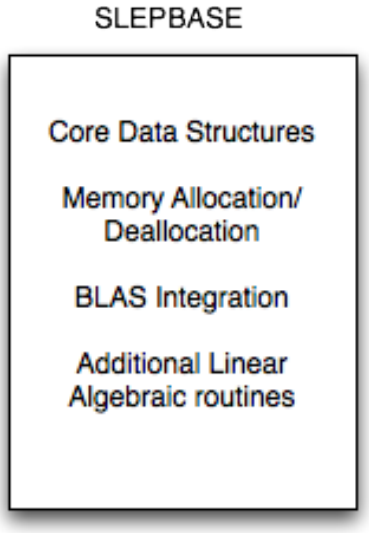
Figure 4.2: SLEP UTILS module - utility functions



The Base module, is the key element of the SLEP package. The Basic Linear Algebra Subroutines (BLAS) is a standard in numerical computation which defines many linear algebraic algorithms and their data structure. Netlib [] defines the standard for the BLAS routines, which are built to be efficient, portable and widely available. The SLEP package provides a default implementation of a subset of the BLAS routines, and also provides options to use existing implementations

such as ATLAS, AMD ACML or Intel MKL instead of the default one. Also, the SLEP package is built around a few data structures which will be discussed later in this section. One of the important things in this package is the memory allocation routines, which are used throughout the other algorithms. The allocation and de-allocation also involves setting up the data structures and destroying data structures as demanded by the request.

Figure 4.3: SLEP BASE module - core data structures and linear algebra routines



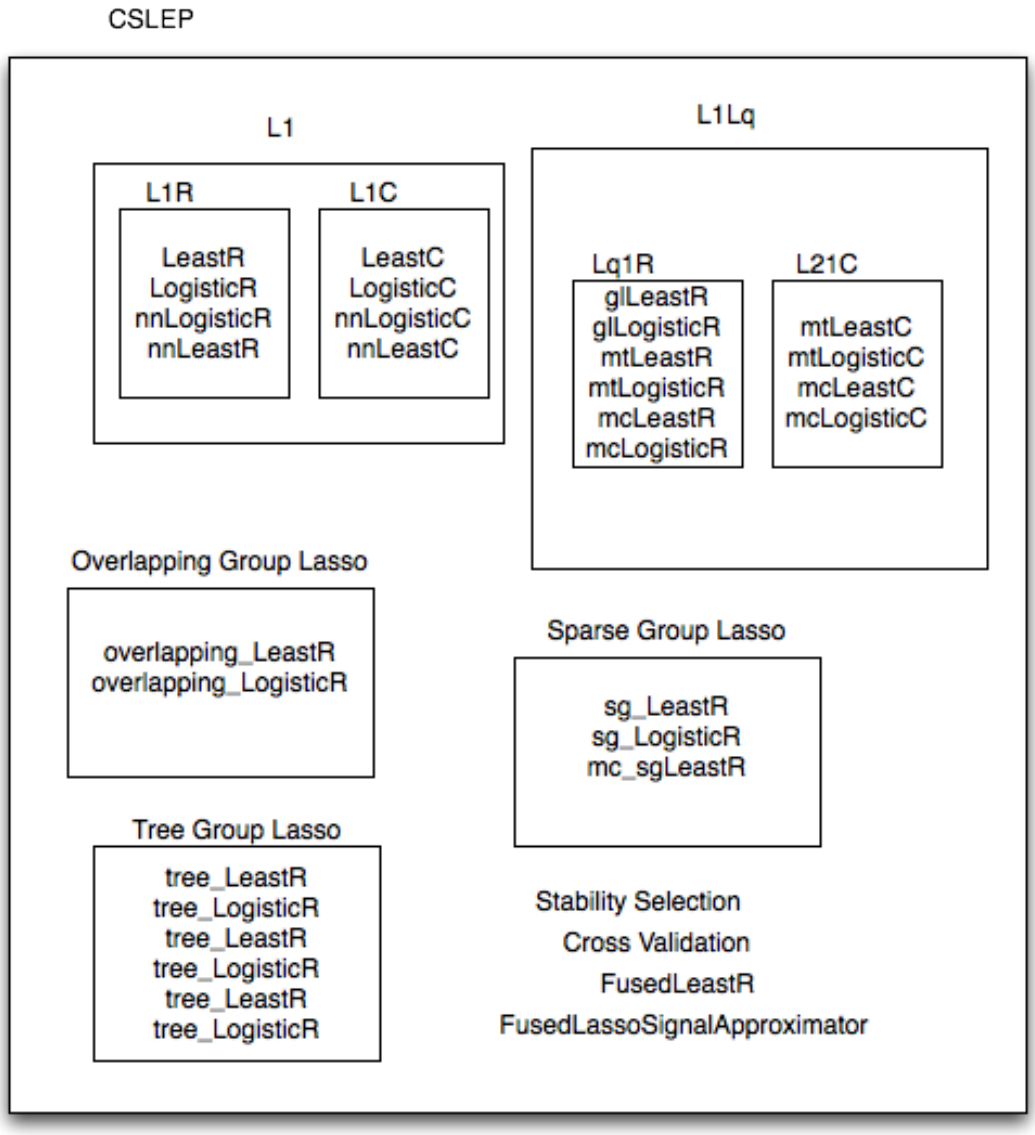
Sparse Learning Algorithms

The sparse learning methods have already been discussed in chapters, *Introduction* and *Forumulation and Algorithms*. The following is a list of the available methods. The following diagram lists all the available methods and their organization internally. Here L1 represents the ℓ_1-norm related algorithms, L1Lq represents ℓ_1/ℓ_q norm related algorithms, the C and R stand for Constrained and Regularized versions of the problems, since we can reformulate the constraints to regularizer terms and vice versa.

The following figure gives a explanation of the various sparse learning routines available in the package and their organization. This forms the crux of the

sparse learning package, and they use the iterative schemes discussed in previous sections.

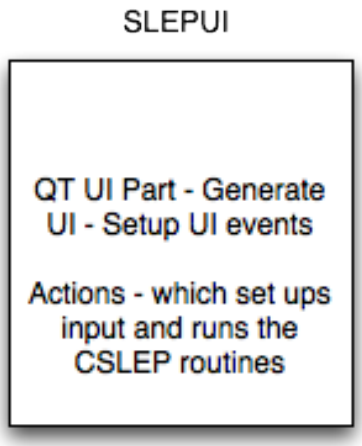
Figure 4.4: CSLEP module - sparse learning algorithms



UI and Visualization

The UI was completely written in Qt for the package. This makes the package cross-platform and helps for a uniform environment in the three mainly available OS, Windows, Mac OS X and Linux. The package calls routines from the other modules, for time measurement, allocation, creation and generation (random) of data, as well as calling sparse learning methods, testing the methods, and running batch algorithms. The UI provides for the user to interact and provide customizable input for their choice of sparse learning method, and export the results to files. The visualization is provided to view how sparse the data is, and the convergence of the iterations, and the path-wise solutions.

Figure 4.5: SLEP UI module - front end

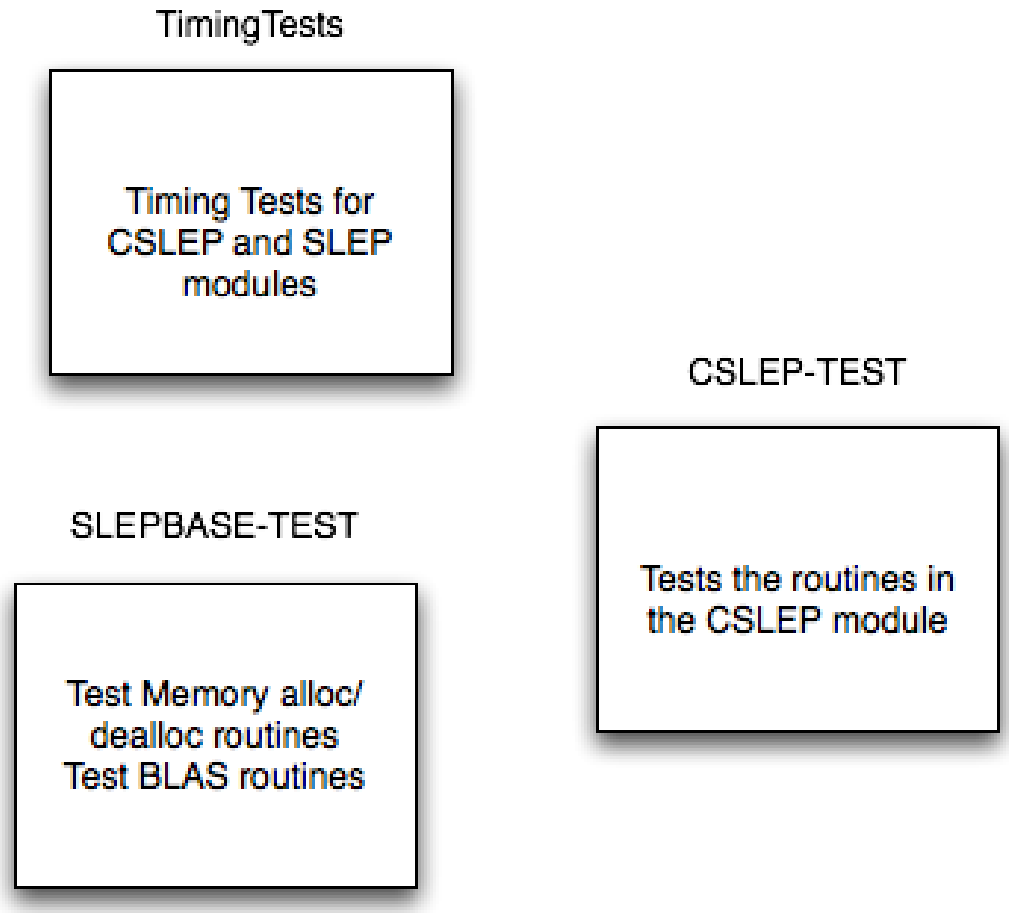


Test modules

The packages described above need to be checked for their correctness and efficiency as this is the key purpose of the SLEP package. Also they can be subject to change in the future, and the changes may introduce errors or bottlenecks. Hence we implement the test package along with the SLEP package to ensure its correct-

ness and efficiency. The `TimingTests` gives us information about the time taken to run the sparse learning and BLAS algorithms. The `SLEPBASE-TEST` tests the basic datastructures, the memory allocation/ deallocation routines, and the BLAS subroutines, which might even be from an external framework. The `CSLEP-TEST` tests the sparse learning algorithms for correctness, and also has some example routines for reference.

Figure 4.6: SLEP Tests module



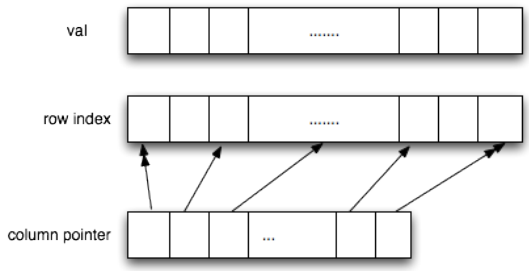
Data structures

The main data structures used in the package are the `slep_matrix` and `slep_vector`. The `slep_matrix` internally can store dense and sparse matrices, and has provisions for future additions of algorithms supporting symmetric, banded, and trian-

gular matrices. While the current package does not demand for the necessity of such types of matrices, future additions of new algorithms may require such special cases. Their design also makes use of consistent and compact storage of values, without allowing redundancy. A further substructure in the `slep_matrix` and `slep_vector`, is the `slep_sparse_matrix` and `slep_sparse_vector`. The `slep_sparse_matrix` allows for storage of anything other than a dense matrix. And similarly `slep_sparse_vector` is used to store sparse vectors.

The sparse matrices are stored in an efficient format called the compressed sparse format. Based on which dimension it is stored it is further divided into compressed sparse row format and compressed sparse column format. This way we can store sparse matrices in an efficient manner. For example, a sparse matrix of dimension 1000×5000 having just 10% of the entries can be stored with $500000 \times 2 + 5000$ space compared to dense matrices which store all the 5000000 elements which is almost 10 times more wasted storage for those elements which are zeros. If the percentage of non-zeros are even less, then there is even more wasted storage in dense matrix. The following diagram illustrates the storage of the elements in compressed sparse column storage format.

Figure 4.7: Compressed Column Storage



The column pointer points to the index in the row index, where each column starts, and hence the last value of the column pointer, gives the total number of non-zero elements present in the sparse matrix. The row index, and the values,

correspond to each other, as we can see, the number of storage required is much less than storing the entire dense matrix.

4.2 Numerical routines for Linear Algebra

The BLAS or Basic Linear Algebraic Subroutines, are a set of standards defined by netlib [**?**], for efficient linear algebraic problems. The core of the SLEP package has a lot of linear algebraic computations. There are 3 levels of BLAS based on the type of operation and the operators involved. The Level 1 BLAS, consists of scalar, vector and vector-vector operations. Level 2 BLAS consists of matrix-vector operations. The SLEP package rigorously uses these 2 levels of BLAS routines for computations. The Level 3 BLAS performs matrix-matrix computations and even triangular matrix solutions. For this, we implemented some of the BLAS algorithms in C, and for further speed up there is option to compile the programs with ATLAS, AMD'S ACML or INTEL MKL. These are known to be the fastest linear algebraic routines available, and some even utilizes multi-threading capabilities of the processor to parallelize the code. The ATLAS (Automatically Tuned Linear Algebra Subroutines) provide a fast running linear algebra library on many platforms. The SLEP code makes use of this, and uses ATLAS in Linux and Windows.

vecLib in Mac OS X

The vecLib in Mac OS X is a library which includes many fast math routines. Specifically it has support for blas, and its implementation is optimized for the Mac OS X. Our software makes use of this framework. This allows for fast execution times of algorithms as the numerical routines are optimized for the platform. Apart from the usual BLAS routines vecLib framework has a few more functionality, which are quite useful and efficient in linear algebraic expressions.

4.3 Matlab integration

The package supports functionality to include the implemented functions in matlab, by allowing mex files to call the C funtions without any change in most of the cases or with little change. Apart from these, the package can read and write .mat files which contain vector and array data types. Allowing the tool to read matlab files, broadens its scope and it can be used without the hassles of writing code in matlab for the various algorithms, batch algorithms, and for commonly used functionality.

The appendix gives a detailed information about the package. There is a section provided for header files, their use, and another provides sample code for executing `fusedLeastR`. There is a detailed walkthrough of the whole package, step by step on how to use the UI to real datasets. Please refer to APPENDIX : *Header Files*, *Example code to run FusedLeastR*, and *Walkthrough of the SLEP Software*.

Chapter 5

DATASETS, RESULTS AND INTERPRETATION

5.1 Timing tests

The purpose of the SLEP package was to have a native running software/library with fast sparse learning routines. Developing it in C and C++ has many advantages compared to Matlab. The most important factor being speed. The timing of the various routines in their matlab prototype vs SLEP Package is shown in the figure. The running times are at least one third faster than the matlab counter parts, with low dimensional data having additional speed ups. The speed up is attributed to native implementation of the algorithms and availability of fast linear algebraic routines.

Figure 5.1: Timing Tests

		100x100		*500x500*		*1000x1000*		*100x10000*		*10000x10000*	
		Matlab Prototype	C	Matlab Prototype	C	Matlab Prototype	C	Matlab Prototype	C	Matlab Prototype	C
LeastR	m=0,l=0	0.173	0.010	0.135	0.105	1.173	0.689	4.921	2.908	87.623	62.416
	m=1,l=0	0.151	0.012	0.129	0.113	1.343	0.720	4.819	3.283	88.826	65.224
	m=1,l=1	0.088	0.006	0.123	0.083	0.849	0.469	5.765	3.501	66.418	42.836
nnLeastR		0.148	0.009	0.103	0.096	0.998	0.516	4.570	2.874	89.038	53.452
LogisticR	m=0,l=0	0.110	0.010	0.215	0.116	0.690	0.341	3.596	3.008	47.828	37.957
	m=1,l=0	0.127	0.015	0.241	0.144	0.869	0.450	5.334	3.183	48.378	26.865
	m=1,l=1	0.055	0.012	0.145	0.100	0.648	0.325	2.886	1.551	47.129	27.397
nnLogisticR		0.163	0.010	0.126	0.068	0.939	0.443	5.742	2.954	38.763	21.150
LeastC	l=0	0.246	0.036	0.201	0.195	0.627	0.976	4.643	3.029	26.027	17.407
	l=1	0.205	0.043	0.109	0.086	0.552	0.329	0.787	0.490	25.813	14.152
nnLeastC		0.121	0.021	0.202	0.078	0.555	0.283	1.719	1.228	24.738	14.487
LogisticC	l=0	0.354	0.053	0.703	0.631	4.557	2.997	4.978	4.649	153.183	109.532
	l=1	0.476	0.066	0.902	0.696	4.826	3.437	1.272	0.973	111.416	77.936
nnLogisticC		0.324	0.040	1.125	0.592	3.949	2.316	5.413	4.741	109.525	67.780
FusedLeastR		0.287	0.073	0.580	0.782	4.876	3.620	9.516	9.181	363.956	348.721
FusedLogisticR		0.377	0.088	1.138	0.886	4.645	3.923	7.904	7.579	353.618	341.716

The sparse learning routines have been implemented with many of the BLAS routines, and there is a possibility of optimizing the expressions. The next versions of SLEP will focus on optimized expressions, which will result in even faster running algorithms. Despite not being optimized at code level, the SLEP package performs better than the Matlab prototype. There are further possiblities of using parallel and distributed implementations of the linear algebraic routines which can

scale the speeds even further.

5.2 AD Dataset

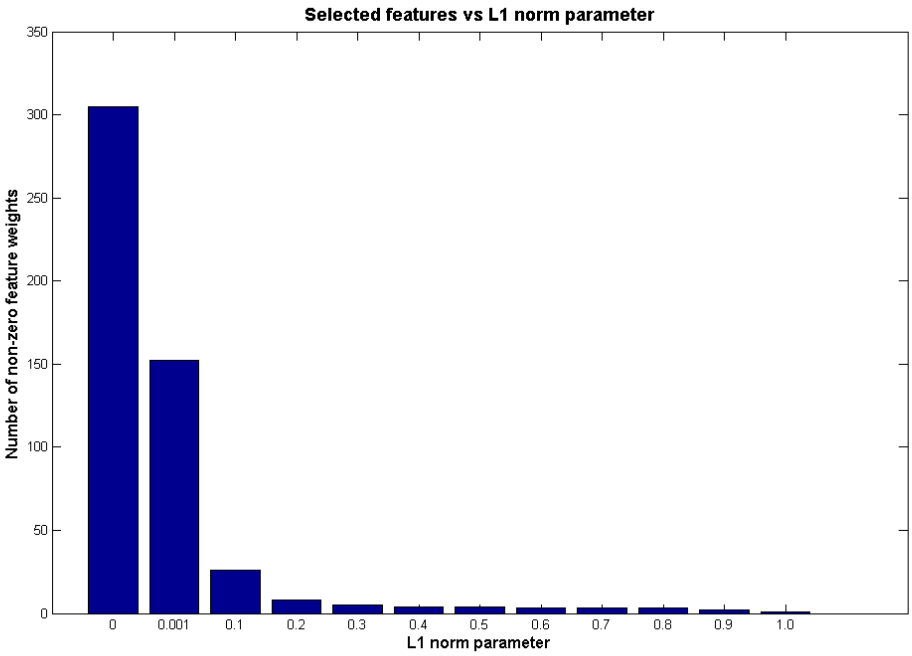
The AD dataset was originally a collection of files in different formats spread across several folders. The refined data, ADNL, consists of 329 records, and 305 features.

5.3 Stability Selection on AD data

For evaluating the effectiveness of the Stability Selection, we run stability selection on the AD data, and get the probability of the features. This is done as follows, for the ℓ_1-norm parameter from 0.025, incremented by 0.025 up until 0.3, we run stability selection 10000 times. This gives the stability paths, which are the probabilities that the feature occured for different parameter values. The stability selection as discussed earlier takes the maximum probability in the stability path ignoring the remaining probability for that feature, and this will correspond to one of the parameter value which is not important. Hence we get a vector containing probabilities for the features. This can be thought of as the ranking of the features, with higher ones being better than the ones with low probabilities. As with any simulations, the larger the number of times stability selection is run, the better the probabilities we get are. Hence in this, we run it for 10000 times. One advantage of the SLEP package is that, it can be many times faster than the matlab prototype for the same implementation. This gives us the power to run it for large number of times, getting relatively more accurate probabilities and hence ranking of features. Also the parameter which induces sparsity over which the stability paths are drawn, should have a relatively smooth path over the probabilities. But if we see that as we increase the parameter, many features are not selected after a particular value, it is better to run the algorithm only until that value, as it is not necessary to reduce the parameter even further. We do not use the whole range allowed for the ℓ_1 norm, which is from 0 to 1, since the increase in the coefficient reduces the num-

ber of non-zeros drastically with ℓ_1-norm parameter equals 1 having lesser than 5 nonzero entries. The following graph shows the number of non-zero features as λ_1 is increased.

Figure 5.2: Selected features



The stability paths as a whole determine the inclusion of a feature, as a feature having the maximum probability of occuring for a particular value of the sparsity inducing parameter, does occur for other values of the parameter as well. The features that have very less probabilities, and many tend to zeros for most cases.

The stability paths of the top 10 features tend to be more smooth. Though the maximum probability can occur anywhere, most among the 10 features have the maximum occuring at 0.3.

The following figure shows all the stability paths for all the 305 features plotted together. The top 10 features are colored blue while the bottom 10 features are colored red, the rest are black.

Figure 5.3: Top 10 features - Stability Paths

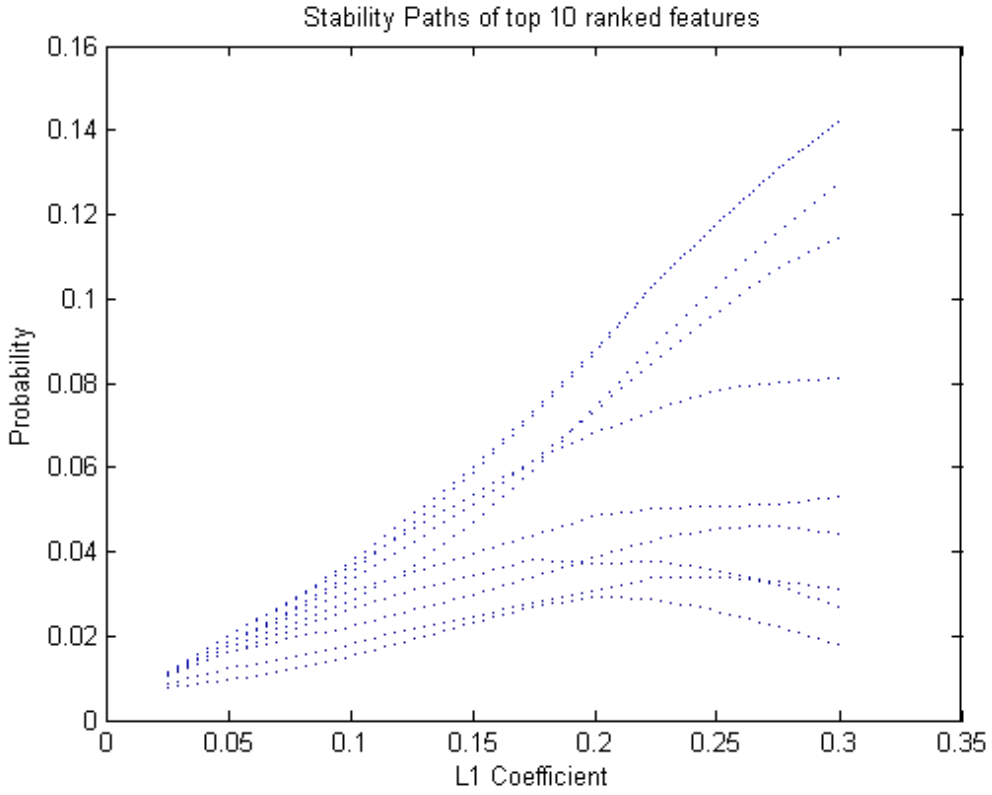
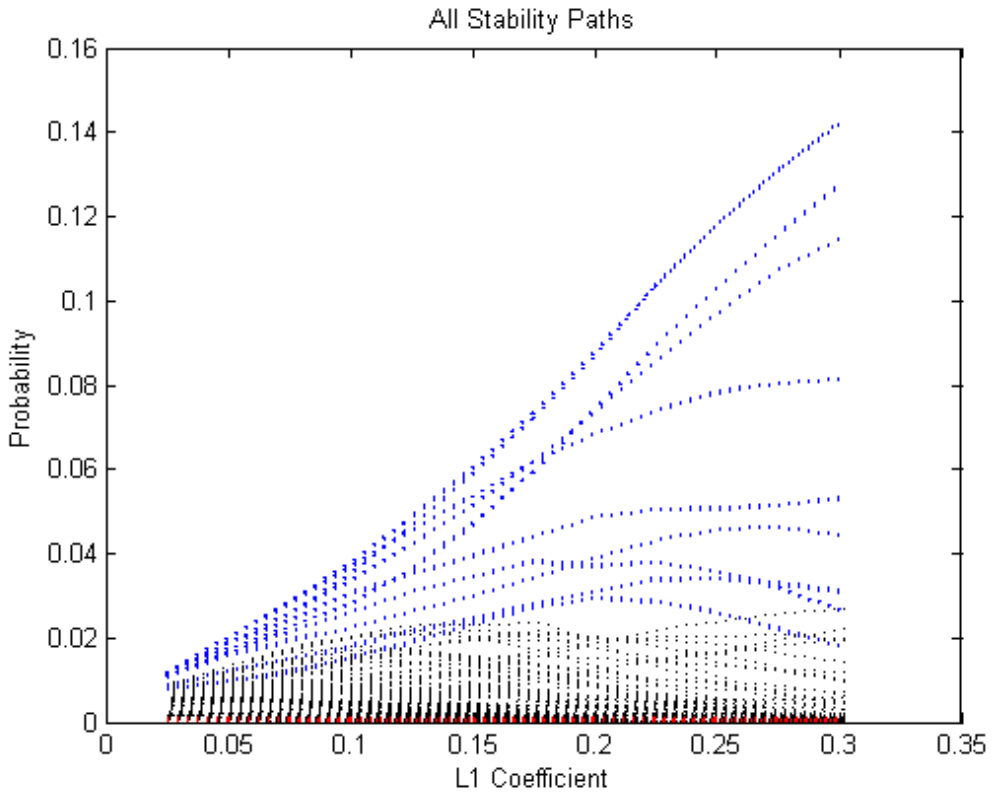
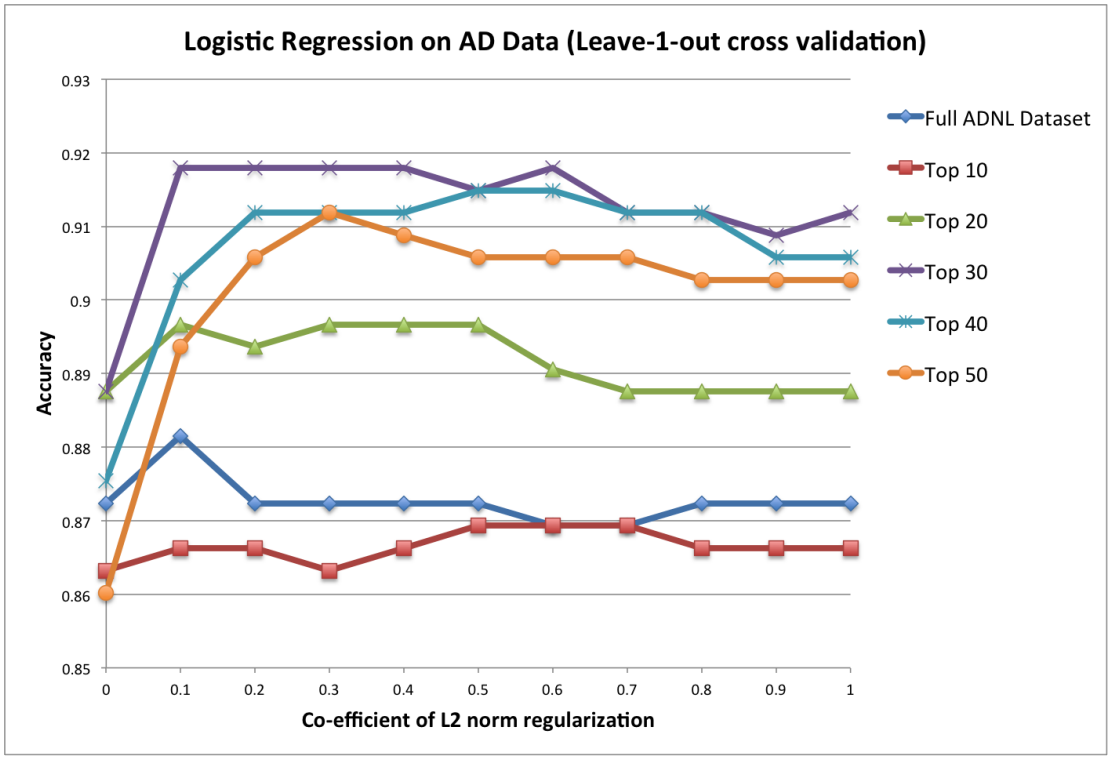


Figure 5.4: Stability Paths of all features



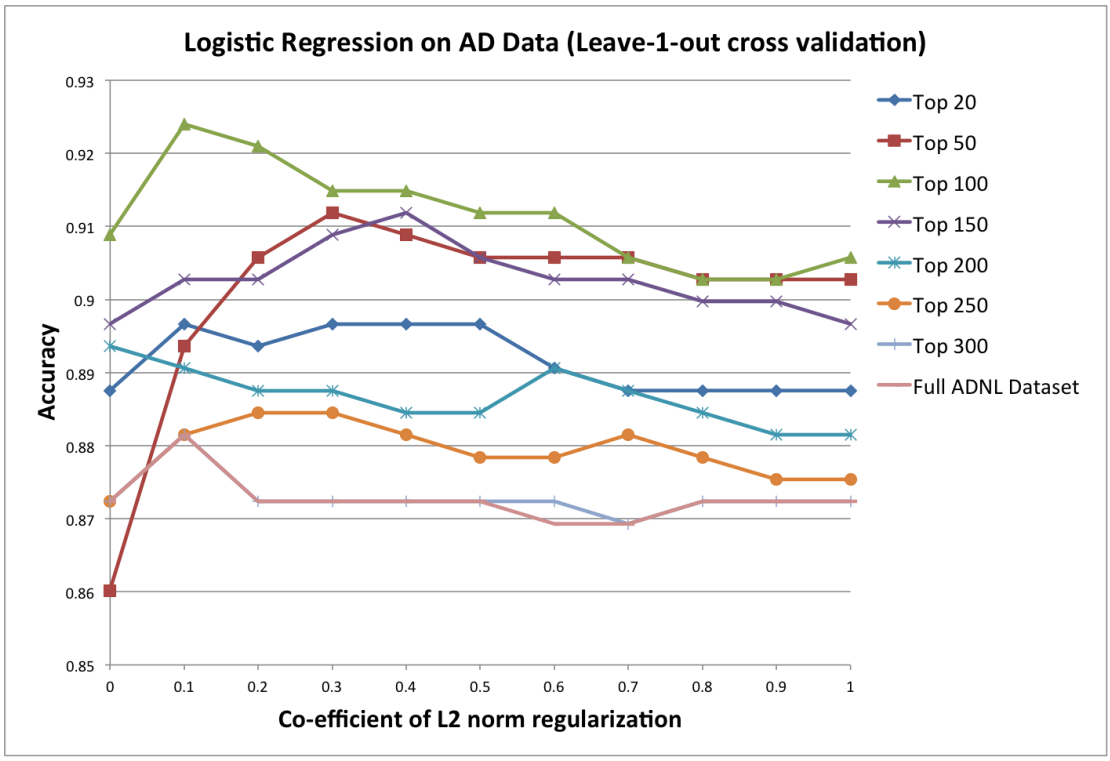
The leave-1-out cross validation, trains and tests as many times as the number of samples, each time testing a new sample while the rest were used to train. It is a good measure of accuracy of how well a particular algorithm performs for a given dataset. We run leave-1-out cross validation for different number of features. The Logistic Regression algorithm is used to train and test with the reduced data. Note that the Logistic Regression algorithm does not contain ℓ_1 norm regularization term but has ℓ_2-norm regularization term. The run for upto top 50 features selected and the accuracy scores for different ℓ_2 norm coefficient is shown in the image below.

Figure 5.5: Accuracy for upto top 50 features selected



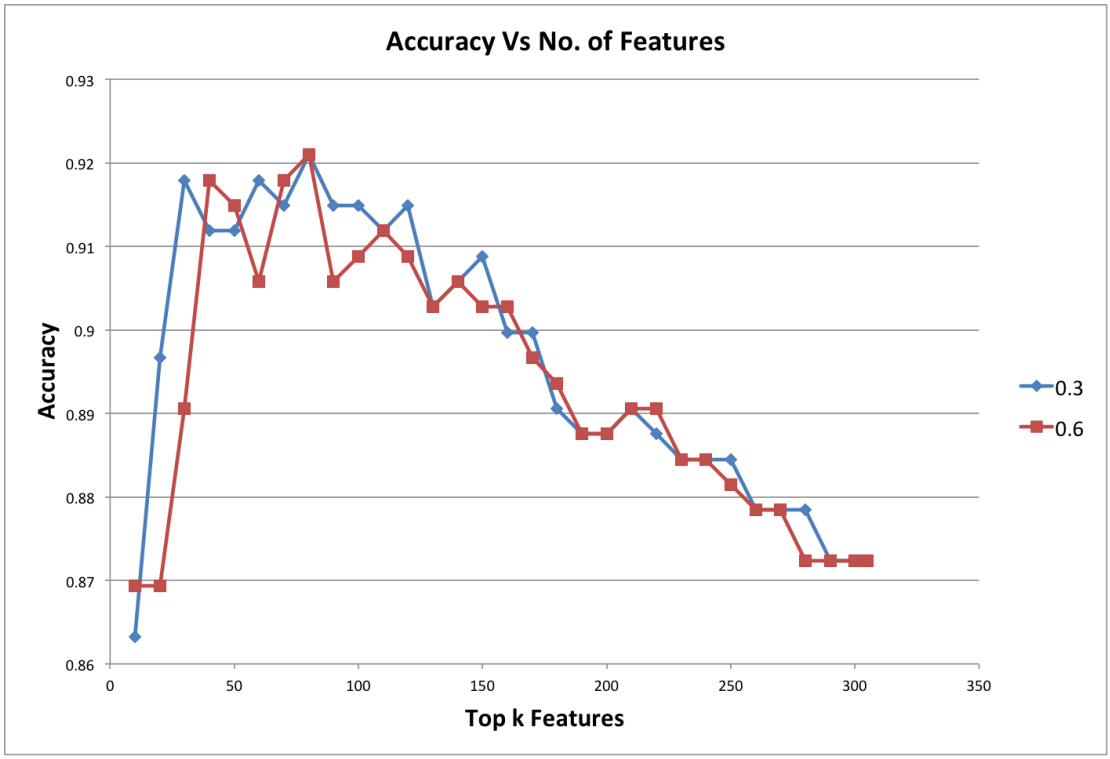
The following figure shows a more comprehensive selection of features. We can see that the algorithm performs better when the number of features are approximately 70.

Figure 5.6: Accuracy for upto all features selected



Also the accuracy scores for 2 particular values of the ℓ_2 norm coefficient over different number of features selected is shown below.

Figure 5.7: Accuracies for ℓ_2 parameters 0.3 and 0.6



Chapter 6

CONCLUSION AND FUTURE WORK

The sparse learning package has a potential to be a vital tool for scientists, to remove irrelevant information, or as a dimensionality reduction technique. Its use of the latest algorithms with the best convergence rates, allow its use in a large-scale setting, and the performance is better than a matlab prototype. Also, the implementation of it as an independent tool, now gives way to a broader scope of audience, who do not use matlab or other tools for their research, to benefit from this software. The simple but useful visualizations allow for a fast view of the results. We aim to add more routines to the software in the future, and improve the efficiency even further. Most importantly to extend the number of algorithms to include trace norm minimization algorithms, and the newer algorithms on sparse learning. As discussed earlier, there is scope for improving the efficiency of the algorithms by optimizing the expressions involving vectors and matrices for the fastest execution time. For example, the vecLib in mac contains additional BLAS routines like the `daxpby`, which does $y = \alpha x + \beta y$, in addition to the usual `daxpy` which calculates $y = \alpha x + y$. Such optimizations will lead to a slightly more efficient package. The existing tool utilizes multithreaded capabilities of the system based on the library being used, but the parallelism can also be achieved by using distributed and multicore and graphic processors, which will linearly speed up the algorithms. The nVidia CUDA and OpenCL are some of the specifications which have parallelized BLAS routines. Intel has also released a math library which can use the power of graphic processors for executing math routines. Using such powerful libraries can drastically improve the performance of the current SLEP. The robust design of the SLEP package allows for such enhancements and switch overs in the future.

The algorithms, especially the ones which involve group sparsity, are find-

ing interesting applications in different fields. Adopting sparse learning algorithms and using them for sparse learning follows certain steps: identifying the problem, formulating them as a machine learning problem, understanding the sparse learning theory, or just experimenting the data with sparse learning techniques. Next steps would be to identify features, the relationships among the features, establishing groups, and relationships among the groups, and choosing a particular sparse learning setting to experiment with. Though the sparse learning package provides tools for an intermediate user, the initial steps has to be done by the user to adopt the problem to specific settings. A possible future directions would be to help integrate some of these in the sparse learning package, such as allowing more data manipulation options and also automation as much as the user wishes. The batch processing algorithms, which are important for the sparse learning package help make real decisions, robustly reduce the amount of available data(using stability selection) and check for the effectiveness of solutions and parameters(cross validation). The stability selection, is a breakthrough idea in feature selection and the theoretical and practical guarantees it provides will be put to use in many fields. The cross validation, support for different options executing each of the sparse learning routines, and useful data manipulation controls makes the tool quite important among scientists and engineers. The use of sparse learning in Alzheimer's Disease has shown us that it has good practical uses, and can be vital in many fields including bio-informatics, web mining, signal processing, so on. The key contribution of this thesis is in the development of this tool, and applying it to the AD data. In the future, we plan to focus on the following

- Add many of the existing sparse learning techniques to the package
- Find, solve and develop more sparse learning techniques, by identifying them from existing problems

- Improve the efficiency even further, and integrate advanced powerful libraries which are capable of harnessing distributed and parallel computing technologies
- Application of sparse learning techniques to problems involving high-dimensional data, in fields like genomics, neuroscience, signal processing and disease diagnosis

[1] A. Ahmed and E.P. Xing. Recovering time-varying networks of dependencies in social and biological studies. *Proceedings of the National Academy of Sciences*, 106(29):11878, 2009.

[2] A. Argyriou, T. Evgeniou, and M. Pontil. Convex multi-task feature learning. *Machine Learning*, 73(3):243–272, 2008.

[3] F.R. Bach. Bolasso: model consistent Lasso estimation through the bootstrap. In *Proceedings of the 25th international conference on Machine learning*, pages 33–40. ACM, 2008.

[4] L.S. Blackford, A. Petitet, R. Pozo, K. Remington, R.C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[5] J.F. Bonnans. *Numerical optimization: theoretical and practical aspects*. Springer-Verlag New York Inc, 2006.

[6] S.P. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge Univ Pr, 2004.

[7] E.J. Candès and M.B. Wakin. An introduction to compressive sampling. *Signal Processing Magazine, IEEE*, 25(2):21–30, 2008.

[8] J. Chen, L. Tang, J. Liu, and J. Ye. A convex formulation for learning shared structures from multiple tasks. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 137–144. ACM, 2009.

[9] J.J. Dongarra, J. Du Croz, S. Hammarling, and I.S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[10] J. Duchi, S. Shalev-Shwartz, Y. Singer, and T. Chandra. Efficient projections onto the l1-ball for learning in high dimensions. In *Proceedings of the 25th international conference on Machine learning*, pages 272–279. ACM, 2008.

[11] B. Efron and G. Gong. A leisurely look at the bootstrap, the jackknife, and cross-validation. *The American Statistician*, 37(1):36–48, 1983.

[12] J. Friedman, T. Hastie, H. Hofling, and R. Tibshirani. Pathwise coordinate optimization. *The Annals of Applied Statistics*, 1(2):302–332, 2007.

[13] J. Friedman, T. Hastie, and R. Tibshirani. A note on the group lasso and a sparse group lasso. *Arxiv preprint arXiv:1001.0736*, 2010.

[14] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik. Gene selection for cancer classification using support vector machines. *Machine learning*, 46(1):389–422, 2002.

[15] L. Jacob, G. Obozinski, and J.P. Vert. Group lasso with overlap and graph lasso. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 433–440. ACM, 2009.

[16] A. Kalousis, J. Prados, and M. Hilario. Stability of feature selection algorithms: a study on high-dimensional spaces. *Knowledge and information systems*, 12(1):95–116, 2007.

[17] S.J. Kim, K. Koh, M. Lustig, S. Boyd, and D. Gorinevsky. An interior-point method for large-scale l1-regularized least squares. *Selected Topics in Signal Processing, IEEE Journal of*, 1(4):606–617, 2007.

[18] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International joint Conference on artificial intelligence*, volume 14, pages 1137–1145. Citeseer, 1995.

[19] C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[20] S.I. Lee, H. Lee, P. Abbeel, and A.Y. Ng. Efficient L1 Regularized Logistic Regression. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 21, page 401. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[21] C. Lemaréchal and C. Sagastizábal. Practical aspects of the Moreau-Yosida regularization I: theoretical properties. *RAPPORT DE RECHERCHE-INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE*, 1994.

[22] J. Liu, J. Chen, and J. Ye. Large-scale sparse logistic regression. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 547–556. ACM, 2009.

[23] J. Liu, S. Ji, and J. Ye. Multi-task feature learning via efficient l 2, 1-norm minimization. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pages 339–348. AUAI Press, 2009.

[24] J. Liu, S. Ji, and J. Ye. SLEP: Sparse Learning with Efficient Projections. *Arizona State University*, 2009.

[25] J. Liu and J. Ye. Efficient euclidean projections in linear time. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 657–664. ACM, 2009.

[26] J. Liu, L. Yuan, and J. Ye. An efficient algorithm for a class of fused lasso problems. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2010.

[27] Jun Liu and Jieping Ye. Efficient l1/lq norm regularization. *CoRR*, abs/1009.4766, 2010.

[28] Jun Liu and Jieping Ye. Fast overlapping group lasso. *CoRR*, abs/1009.0306, 2010.

[29] Jun Liu and Jieping Ye. Moreau-yosida regularization for grouped tree structure learning. In J. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R.S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 1459–1467. 2010.

[30] L. Meier, S. Van De Geer, and P. Buhlmann. The group lasso for logistic regression. *group*, 70(Part 1):53–71, 2008.

[31] N. Meinshausen and P. Buhlmann. Stability selection. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(4):417–473, 2010.

[32] A. Nemirovski. Efficient methods in convex programming. 2005.

[33] Y. Nesterov. A method of solving a convex programming problem with convergence rate O (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376, 1983.

[34] Y. Nesterov and I.U.E. Nesterov. *Introductory lectures on convex optimization: A basic course*. Springer Netherlands, 2004.

[35] W.H. Press, B.P. Flannery, S.A. Teukolsky, W.T. Vetterling, et al. *Numerical recipes*. Cambridge Univ. Pr., 1986.

[36] S. Shalev-Shwartz and N. Srebro. Low l1-norm and guarantees on sparsifiability. In *Sparse Optimization and Variable Selection, Joint ICML/COLT/UAI Workshop*. Citeseer, 2008.

[37] J. Shao. Linear model selection by cross-validation. *Journal of the American Statistical Association*, pages 486–494, 1993.

[38] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996.

[39] R. Tibshirani, M. Saunders, S. Rosset, J. Zhu, and K. Knight. Sparsity and smoothness via the fused lasso. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(1):91–108, 2005.

[40] R. Tibshirani and P. Wang. Spatial smoothing and hot spot detection for CGH data using the fused lasso. *Biostatistics*, 9(1):18, 2008.

[41] R.C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[42] M. Yuan and Y. Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006.

[43] P. Zhao, G. Rocha, and B. Yu. Grouped and hierarchical model selection through composite absolute penalties. *Annals of Statistics*, 37(6A):3468–3497, 2009.

[44] P. Zhao, G. Rocha, and B. Yu. The composite absolute penalties family for grouped and hierarchical variable selection. *The Annals of Statistics*, 37(6A):3468–3497, 2009.

Appendix A

Data Structures and BLAS Routines

We explain the different data structures available in the SLEP package for: matrix (dense, sparse and others) and vector (dense and sparse), and also list some of the linear algebra routines that are commonly used in the package.

The SLEP package follows many of the BLAS specifications, modified to fit the data structures used in the SLEP package. Hence for specifying right and left multiplication of a matrix, we have the following:

```
/* Enumerator for Left or Right multiplication of two matrices A,B
    Left : A*B
    Right: B*A
*/
typedef enum{
    SlepLeft, SlepRight
}slep_side;
```

The sparse matrices may be represented in three different formats: Compressed Sparse Row, Compressed Sparse Column and I-J-V formats. As previously discussed, this saves a lot of space if the matrix is large and sparse. The matrix can either be dense, sparse, symmetric, triangular or banded.

```
/* Enumerator for sparse matrix storage format.
 *  Currently 2 are supported Sparse Column and Sparse Row.
**/
typedef enum{
    SlepCompressedSparseColumn, SlepCompressedSparseRow
}slep_sparse_format;

/*
 *  Enumerator for the type of sparsity of the matrix.
 *  5 types of supported currently
 *  1. Symmetric - Store only the N*[N+1]/2 entries
 *  2. Triangular - Store only one half
 *  3. Banded - Store only the non-zero entries ordered sequentially row-wise
 *  4. Sparse - Store in Compressed Column or Compressed Row storage
 *  5. Dense - Store in dense format
**/
typedef enum{
    SlepSparse, SlepDense, SlepSymmetric, SlepTriangular, SlepBanded
}slep_sparsity;

/* Enumerator to specify Transpose or NoTranspose */
typedef enum{
    SlepTranspose,
    SlepNoTranspose
}slep_transposity;

/* Enumerator for Upper or Lower Triangular and Banded Matrices */
typedef enum{
    SlepUpper, SlepLower, SlepBandedUpper, SlepBandedLower, SlepBandedBoth, ↩
        SlepFull
}slep_uplobanded;

/* The core sparse matrix storage structure
```

```
        Stores values(ordered row-wise[or column-wise]),
        their columns[or rows], and a row[or column] index pointer
        pointing to starting value of each row.
*/
typedef struct{
    double *values;
    int *ColumnsORRows;
    int *Index;
    slep_sparse_format format;
}sparse_matrix_base; // Zero Based Indexing Compressed-Sparse-Row Format

typedef struct{
    union{
        sparse_matrix_base spmtrx;
        double *mtrx;
    }data;

        /* This has value in {SlepDense, SlepSparse, SlepSymmetric,
                              SlepTriangular} */
    slep_sparsity sparsity;

        /* This has value in {SlepUpper, SlepLower, SlepBandedUpper,
                      SlepBandedLower, SlepBandedBoth, SlepFull} */
    slep_uplobanded uplobanded;

    /* M Rows */
    int m;

    /* N Columns */
    int n;

        /* Leading Dimension (used for Dense Matrices and Symmetric or
           Banded Matrices) */
    int ld;

    /* Bandwidth */
    int kl;
    int ku;
}slep_matrix;

/* Structure to represent sparse vectors */
typedef struct{
    double *values;
    int *index;
    int nnz;
}sparse_vector_base;

typedef struct{
    union{
        sparse_vector_base spvctr;
        double *vctr;
    }data;

    /* This has value in {SlepDense, SlepSparse} */
    slep_sparsity sparsity;

    int inc;
    int dim;
}slep_vector;
```

Listed below are some of the commonly used SLEP BLAS routines that are used inside the algorithms.

```
/* [x = alpha * x] */
void slep_dscal(double alpha, slep_vector* x);

/* [y = y + alpha * x] */
void slep_daxpy(double alpha, slep_vector* x, slep_vector* y);

/* [y = alpha * x + beta * y] */
void slep_daxpby(double alpha, slep_vector* x, double beta, slep_vector* y);

/* [= x.y] */
double slep_ddot(slep_vector* x, slep_vector* y);

/* [=sqrt(x.x)] */
double slep_dnrm2(slep_vector* x);

/* [=|x|_p (p-norm of x)] */
double slep_dpnrm(slep_vector* x, double p);

/* [=|x| (1-Norm)] */
double slep_dasum(slep_vector* x);

/* [=sum(x_i) (Sum of all elements of x)] */
double slep_dsum(slep_vector* x);

/* [=i such that |x_i| is maximum ] */
int slep_idamax(slep_vector* x);

/* [=|x_i| such that |x_i| is maximum ] */
double slep_damax(slep_vector* x);

/* [= x_i (such that x_i is maximum) ] */
double slep_dmax(slep_vector* x);

/* Level 2 BLAS */

/* [y = alpha.A`.x + beta.y] */
void slep_daAxpby(slep_transposity trans, double alpha,
                  slep_matrix* A, slep_vector* x,
                  double beta, slep_vector* y);


/* [x = A`.x (` implies SlepTranspose or SlepNoTranspose)]*/
void slep_dtrmv(slep_transposity trans, slep_matrix* A, slep_vector* x);

/* [x = {A`^(-1)} . x (` implies SlepTranspose or SlepNoTranspose)]*/
void slep_dtrsv(slep_transposity trans, slep_matrix* A, slep_vector* x);


void slep_dger(double alpha, slep_vector* x, slep_vector* y,
               slep_matrix* A);
```

Appendix B

Header files

In this section we will describe the available functions for the different sparse learning algorithms and a few of the commonly used functions for allocation and testing in the SLEP package. First we list the *cslep.h* file which contains all the functions that are available in the SLEP package and their definitions.

```
#ifndef CSLEP_H
#define CSLEP_H

#include "slepbase.h"

#ifdef __cplusplus
extern "C" {
#endif

/* L1R */
int LeastR(slep_matrix* A, slep_vector* y, double z, OPTS *opts,
           slep_vector* x, slep_vector* funVal, slep_vector* valueL);

void LogisticR(slep_matrix* A, slep_vector* y, double z, OPTS* opts,
           slep_vector* x, double *c, slep_vector* funVal, slep_vector* ↩
               valueL);

int nnLeastR(slep_matrix* A, slep_vector* y, double z, OPTS *opts,
           slep_vector* x, slep_vector* funVal, slep_vector* valueL);

void nnLogisticR(slep_matrix* A, slep_vector* y, double z, OPTS* opts,
           slep_vector* x, double *c, slep_vector* funVal);

/* L1C */
int LeastC(slep_matrix* A, slep_vector* y, double z, OPTS *opts,
           slep_vector* x, slep_vector* funVal, slep_vector* valueL);

void LogisticC(slep_matrix* A, slep_vector* y, double z, OPTS* opts,
           slep_vector* x, double *c, slep_vector* funVal, slep_vector* ↩
               valueL);

int nnLeastC(slep_matrix* A, slep_vector* y, double z, OPTS *opts,
           slep_vector* x, slep_vector* funVal);

void nnLogisticC(slep_matrix* A, slep_vector* y, double z, OPTS* opts,
           slep_vector* x, double *c, slep_vector* funVal);

/* Lq1R */
void glLeastR(slep_matrix* A, slep_vector* y, double z, OPTS* opts,
           slep_vector* x, slep_vector* funVal, slep_vector* valueL);

void glLogisticR(slep_matrix* A, slep_vector* y, double z, OPTS* opts,
           slep_vector* x, double *c, slep_vector* funVal, slep_vector* ↩
               valueL);

void mtLeastR(slep_matrix* A, slep_vector* y, double z, OPTS* opts,
           slep_matrix* X, slep_vector* funVal, slep_vector* valueL);

void mtLogisticR(slep_matrix* A, slep_vector* y, double z, OPTS* opts,
           slep_matrix* X, slep_vector *c_v, slep_vector* funVal, slep_vector↩
               * valueL);

void mcLeastR(slep_matrix* A, slep_matrix* Y, double z, OPTS* opts,
```

```
            slep_matrix* X, slep_vector* funVal, slep_vector* valueL);

void mcLogisticR(slep_matrix* A, slep_matrix* Y, double z, OPTS* opts,
            slep_matrix* X, slep_vector *c_v, slep_vector* funVal, slep_vector↩
                * valueL);

/* L21C */
void mtLeastC(slep_matrix* A, slep_vector* y, double z, OPTS* opts,
            slep_matrix* X, slep_vector* funVal, slep_vector* valueL);

void mtLogisticC(slep_matrix* A, slep_vector* y, double z, OPTS* opts,
            slep_matrix* X, slep_vector *c_v, slep_vector* funVal, slep_vector↩
                * valueL);

void mcLeastC(slep_matrix* A, slep_matrix* Y, double z, OPTS* opts,
            slep_matrix* X, slep_vector* funVal, slep_vector* valueL);

void mcLogisticC(slep_matrix* A, slep_matrix* Y, double z, OPTS* opts,
            slep_matrix* X, slep_vector *c_v, slep_vector* funVal, slep_vector↩
                * valueL);

void flsa(double *x, double *z, double *infor,
          double * v, double *z0,
          double lambda1, double lambda2, int n,
          int maxStep, double tol, int tau, int flag);

int fusedLogisticR(slep_matrix* A, slep_vector* y, double lambda, OPTS *opts,
            slep_vector* x, double *c, slep_vector* funVal, slep_vector* ↩
                valueL);

int fusedLeastR(slep_matrix* A, slep_vector* y, double z, OPTS *opts,
            slep_vector* x, slep_vector* funVal, slep_vector* valueL);

void general_altra(double *x, double *v, int n, double *G, double *ind, int ↩
    nodes);
void general_altra_mt(double *X, double *V, int n, int k, double *G, double *↩
    ind, int nodes);
void altra(double *x, double *v, int n, double *ind, int nodes);
void altra_mt(double *X, double *V, int n, int k, double *ind, int nodes);

int sgLeastR(slep_matrix* A, slep_vector* y, double lambda1, double lambda2, ↩
    OPTS *opts,
            slep_vector* x, slep_vector* funVal, slep_vector* valueL);
int sgLogisticR(slep_matrix* A, slep_vector* y, double lambda1, double lambda2↩
    , OPTS *opts,
            slep_vector* x, double *c, slep_vector* funVal, slep_vector* ↩
                valueL);
int mcsgLeastR(slep_matrix* A, slep_matrix* Y, double lambda1, double lambda2,↩
     OPTS *opts,
            slep_matrix *X, slep_vector* funVal, slep_vector* valueL);

int tree_LeastR(slep_matrix* A, slep_vector* y, double z, OPTS *opts,
            slep_vector* x, slep_vector* funVal, slep_vector* valueL);

int tree_LogisticR(slep_matrix* A, slep_vector* y, double z, OPTS* opts,
            slep_vector* x, double *c, slep_vector* funVal, slep_vector* ↩
                valueL);


int tree_mtLeastR(slep_matrix* A, slep_vector* y, double z, OPTS* opts,
            slep_matrix* X, slep_vector* funVal, slep_vector* valueL);
int tree_mtLogisticR(slep_matrix* A, slep_vector* y, double z, OPTS* opts,
            slep_matrix* X, slep_vector *c_v, slep_vector* funVal, slep_vector↩
                * valueL);
int tree_mcLeastR(slep_matrix* A, slep_matrix* Y, double z, OPTS* opts,
```

```
            slep_matrix* X, slep_vector* funVal, slep_vector* valueL);
int tree_mcLogisticR(slep_matrix* A, slep_matrix* Y, double z, OPTS* opts,
            slep_matrix* X, slep_vector *c_v, slep_vector* funVal, slep_vector
                * valueL);


double LogisticRTest(slep_matrix* A, slep_vector* y, slep_vector* x, double c)
    ;
double LeastRTest(slep_matrix* A, slep_vector* y, slep_vector* x);



/* Stability Selection, and other methods */

int kFoldCV(int k, slep_matrix* A, slep_vector* y, slep_vector *param1,
    slep_vector* param2, OPTS *opts, char* method_name,
            slep_vector* accuracy);

int stabilitySelection(int nRuns, slep_matrix* A, slep_vector* y, slep_vector
    *z, OPTS *opts, char* method_name,
        slep_matrix* reducedDimensionA, slep_vector *xProbabilities );

int leave1OutCV(slep_matrix* A, slep_vector* y, slep_vector *param1,
    slep_vector* param2, OPTS *opts, char* method_name,
            slep_vector* accuracy);

int pathwise(slep_matrix* A, slep_vector* y, slep_vector *z, OPTS *opts,
        slep_matrix* X, slep_vector* funValLambda, char* method_name);

int bolasso(slep_matrix* A, slep_vector* y, double lambda, OPTS *opts, char*
    method_name, int nRuns,
        slep_vector *x, slep_vector* funVal, slep_vector* valueL);

#ifdef __cplusplus
}
#endif

#endif //CSLEP_H
```

Some definitions from the *slep.h* header file

```
slep_vector* slepAllocDenseVector(int size);
slep_vector* slepAllocSparseVector(int nnz);
int slepFreeVector(slep_vector* vctr);
slep_vector* slepCopyToNewVector(slep_vector* vctr);

slep_matrix* slepAllocDenseMatrix(int m,int n);
slep_matrix* slepAllocSparseMatrix(int m,int nnz);
int slepFreeMatrix(slep_matrix* mtrx);
int findNnz(slep_matrix* mtrx);
```

The following are related to the OPTS field which is passed on as an argument to most of the methods of the SLEP package

```
/* Set / Unset the OPTS with a particular field name, if set, this value will ↩
    be used if required, else a new value created */
void setOPTSREGISTER(OPTS *opts,OPTS_REGISTRY field);
void unsetOPTSREGISTER(OPTS *opts,OPTS_REGISTRY field);

/* Check to see if a field has been set in the OPTS */
int hasGot(OPTS *opts,OPTS_REGISTRY field);

/* Initialize OPTS to the default options */
void sll_opts(OPTS* opts);

/* initializes normalization of the A matrix, by creating / required mu and nu↩
     matrices for normalization */
void initNormalization(slep_matrix* A, OPTS* opts);
```

The following routines are used to copy a vector or matrix into another, and to set it to all zeros.

```
void slep_zerom(slep_matrix *mtrx);
void slep_zerov(slep_vector *vctr);
/* [dest = orig] */
int slep_dvcopy(slep_vector* orig, slep_vector* dest);
int slep_dmcopy(slep_matrix* orig, slep_matrix* dest);
slep_matrix* slepCopyToNewMatrix(slep_matrix* mtrx);
slep_vector* slepCopyToNewVector(slep_vector* vctr);
```

The following are defined to get time measurement from the system. It has been customized for each of the OS.

```
#if defined(__APPLE__)
    #include <mach/mach_time.h>
    #include <time.h>
    typedef struct{
        uint64_t start;
        uint64_t stop;
    }Timer;
#elif defined(WIN32)
    #include <windows.h>
    typedef struct {
        LARGE_INTEGER start;
        LARGE_INTEGER stop;
```

```
    }Timer;
    double LIToSecs(LARGE_INTEGER *L);
#else
    #include<time.h>
    typedef struct{
        struct timespec start;
        struct timespec stop;
    }Timer;
#endif

void startTimer(Timer *timer);
void stopTimer(Timer *timer);
double getElapsedTime(Timer *timer);
```

The following routines are used to generate random numbers in the SLEP package.

```
/* generate a single random number from the given normal distribution */
double Normal(double m, double s);

/* Used to generate streams of 'n' random numbers for a given normal ↩
    distribution */
void slep_randn(double* x, int n, double mu, double sigma);
```

Appendix C

Example code to run FusedLeastR

Using the SLEP package in your program and running it from command line was one of the important requirement behind the purpose of SLEP. The following code shows how the FusedLeastR routine is called from command line, compiled and executed. The *main.cpp* file contains the code as below.

```
#include "slepbase.h"
#include "cslep.h"

void exampleFusedLeastR1(){
    /* define all the necessary variables */
    OPTS opts;
    slep_matrix* A;
    slep_vector *x, *y, *funVal, *LValue;
    double rho;
    Timer timer;
    int maxiterations;


    /* Allocate memory for matrices and vectors */
    A = slepAllocDenseMatrix(1000,1000);
    x = slepAllocDenseVector(1000);
    y = slepAllocDenseVector(1000);

    /* Allocate memory for funVal and LValue to hold upto 'maximum iterations'↩
         entries */
    maxiterations = 100;
    funVal = slepAllocDenseVector(maxiterations);
    LValue = slepAllocDenseVector(maxiterations);


    /* Generate synthetic data using the random number routines */
    slep_randn(A->data.mtrx,1000*1000,0.,1.);
    slep_randn(x->data.vctr,1000,0.,1.);
    /* 'y' which will hold the results after generating / will initially ↩
       contain a gaussian noise, which will be added to A*x */
    slep_randn(y->data.vctr,1000,0.,1.);

    /* y = A*x + 0.01* noise */
    slep_daAxpby(SlepNoTranspose,1.,A,x,0.01,y);

    /* no normalization */
    opts.nFlag=0;

    /* l2 regularization coefficient set to zero */
    opts.rsL2=0;
    opts.maxIter=maxiterations;

    /* set init, termination flags and maxIter */
    opts.init = 2;
    opts.tFlag=5;
    opts.maxIter = 100;

    /* rFlag is set to 1 to scale up the regularization coefficient as ↩
       required */
    opts.rFlag = 1;


    /* The l1 regularization constant */
```

```
    rho=0.2;

    /* The fused Penalty term */
    opts.fusedPenalty = 0.1;

    opts.mFlag=0;
    opts.lFlag=0;

    /* Set these in the opts REGISTER */
    setOPTSREGISTER(&opts, SlepOPT_maxIter);
    setOPTSREGISTER(&opts, SlepOPT_rsL2);
    setOPTSREGISTER(&opts, SlepOPT_fusedPenalty);


    /* write the 2 as files*/
    slepWriteDenseMatrix("A.matrix",A);
    slepWriteDenseVector("y.vector",y);


    /* Start the timer */
    startTimer(&timer);

    /* Running fusedLeastR method */
    fusedLeastR(A,y,rho,&opts,x,funVal,LValue);

    /* Stop the timer */
    stopTimer(&timer);

    /* Print the time taken */
    printf("Time taken = %lf\n", getElapsedTime(&timer));

    /* write the results back to file */
    slepWriteDenseVector("x.vector",x);
    slepWriteDenseVector("funVal.vector",funVal);

    /* Free all the allocated memory */
    slepFreeMatrix(A);
    slepFreeVector(y);
    slepFreeVector(x);
    slepFreeVector(funVal);
    slepFreeVector(LValue);


}


int main(){
    /* running the test for Fused Lasso */
    exampleFusedLeastR1();
    return 0;
}
```

As we can see, the two important header files **slepbase.h** and **cslep.h** are required for using the SLEP library in a program. We also need the **CSLEP.lib** library in windows, or libCSLEP.a in Mac and Linux to be statically linked to the program, and compiled and linked. Additionally, preprocessor directives for using ATLAS can be given by defining the keyword (usually specified as the -D keyword option to the compiler) ***ATLAS*** while compiling. If you are using windows, the dynamic library which has the ATLAS implementation should be present while executing the created command line executable.

Appendix D

Walkthrough of the SLEP Software

This section walks through the SLEP software providing step-by-step screenshots and using the SLEP package in everyday setting.

The SLEP software is started by clicking on the `SLEP.exe` file in Windows. It needs the additional *dll* files for various functionality from, UI, Matlab file reading, to core linear algebra routines (ATLAS). Running the executable opens up the following screen.

Figure D.1: SLEP.exe

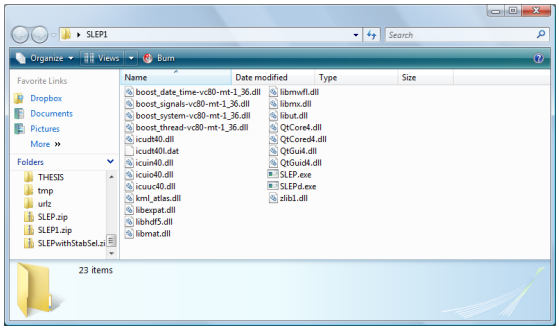
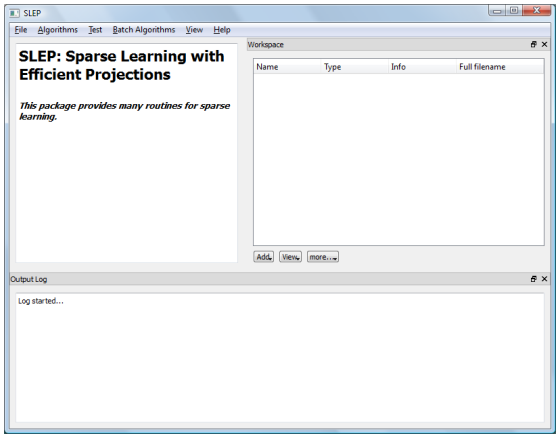


Figure D.2: SLEP Software



THE WORKSPACE

The workspace is the place where all the data currently used and stored by the software is shown. It shows the name, dimensions, type of data (sparse matrix, dense vector,...).

Under the algorithms menu, there are many sparse learning methods as mentioned previously.

Figure D.3: Workspace

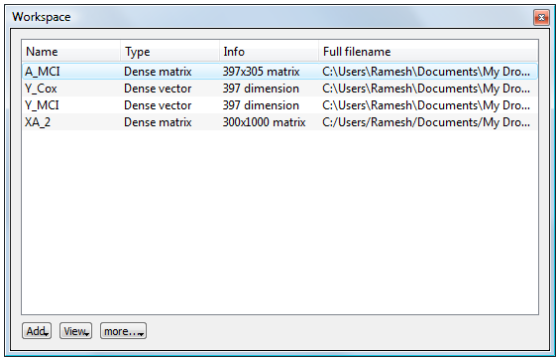


Figure D.4: SLEP Software - Algorithms Menu

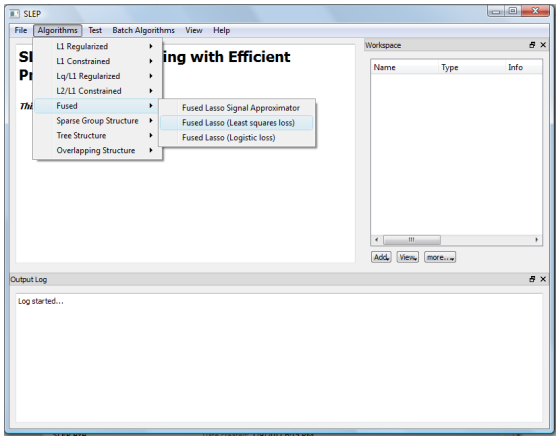
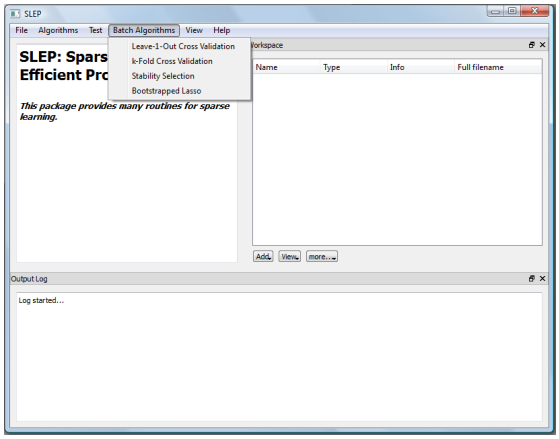


Figure D.5: SLEP Software - Batch Algorithms



We now add the Matlab file containing the AD data into the workspace. This is done by clicking add under workspace, and selecting the `Dataset1.mat` file. Now this imports all the data into the workspace.

Figure D.6: Importing data into workspace

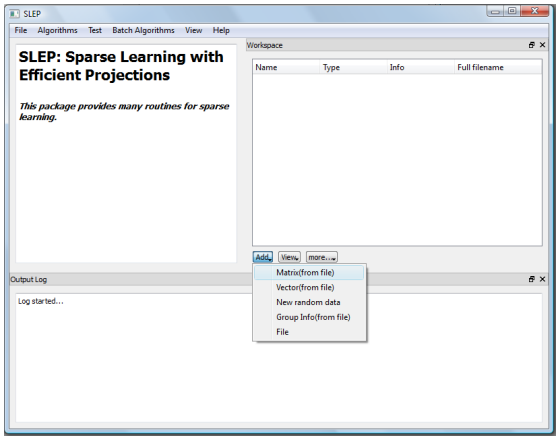
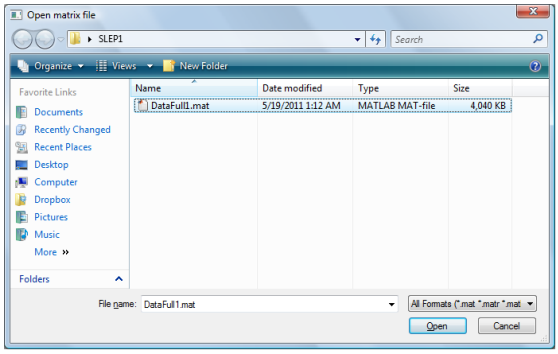
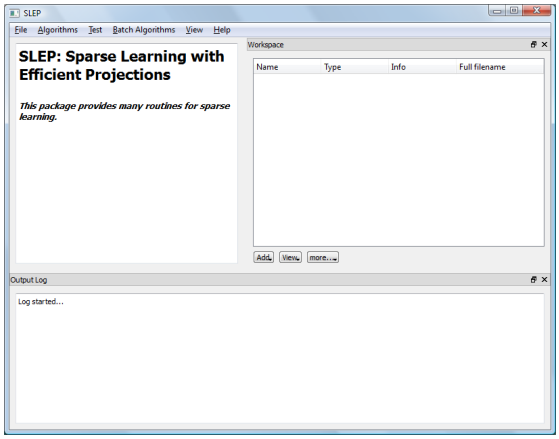


Figure D.7: Importing Data into Workspace 2



Once this is done, we select cross validation bye clicking it from the Batch menu. This opens up the cross validation options dialog, where we need to give the parameter values, the number of folds or if it is a leave-1-out cross validation, and the method to use. We select `LogisticR` here and continue.

Figure D.8: SLEP Software - Cross Validation



Clicking continue shows up more options for running the `LogisticR`. The options menu of the `LogisticR` are shown below.

Figure D.9: SLEP Software - LogisticR options

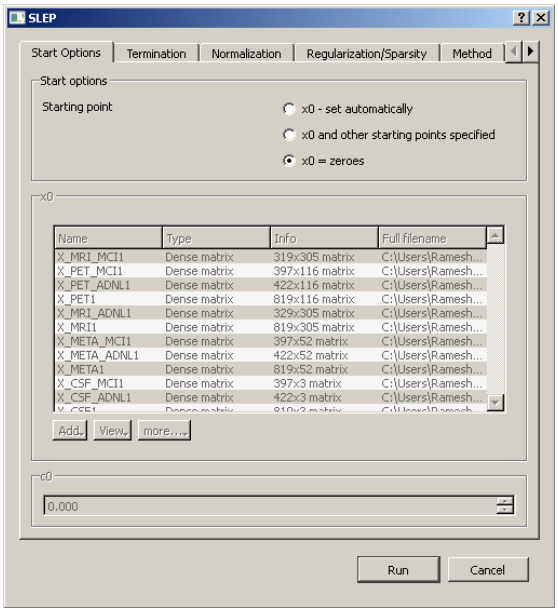
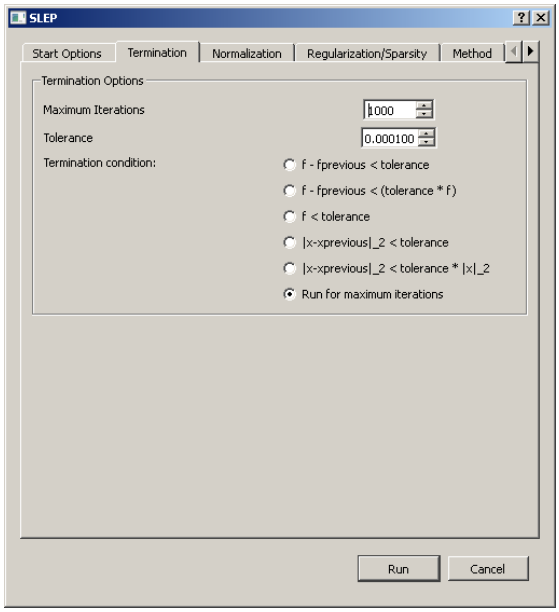


Figure D.10: SLEP Software - LogisticR options



Once the cross validation finishes we get the accuracy for the same.

Figure D.11: SLEP Software - LogisticR options

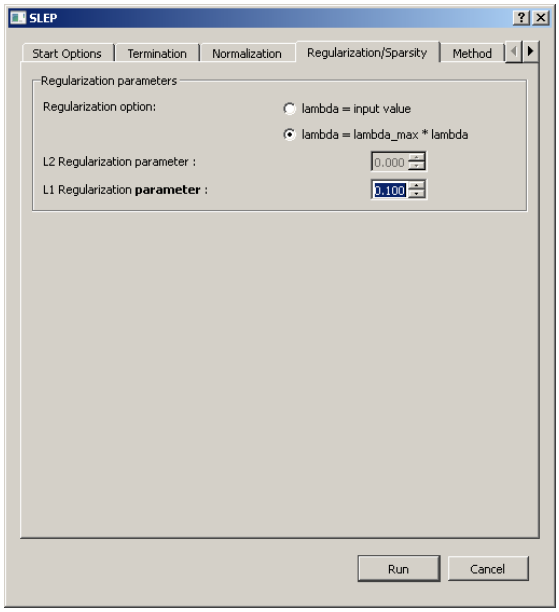


Figure D.12: SLEP Software - LogisticR options

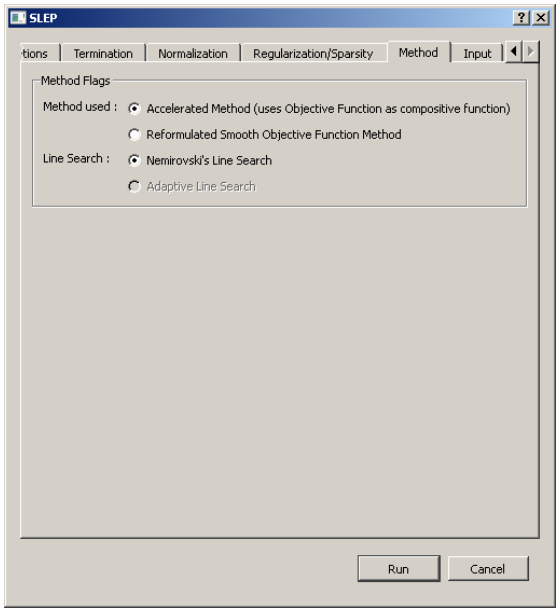


Figure D.13: SLEP Software - LogisticR options

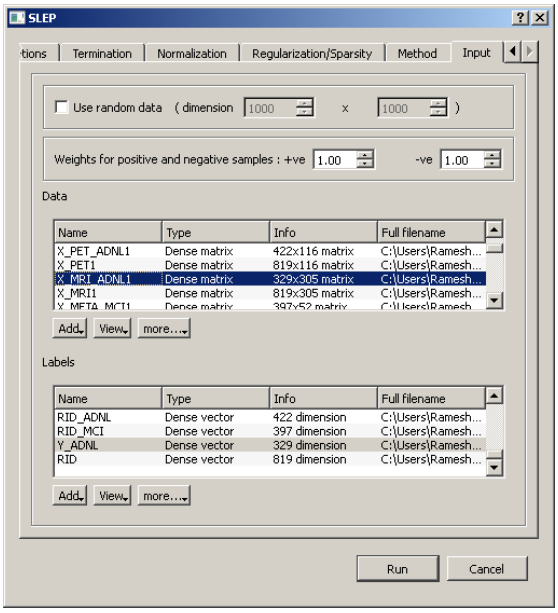


Figure D.14: Cross Validation Accuracy