BOOSTING LIKE PATH ALGORITHMS FOR ℓ_1 REGULARIZED INFINITE DIMENSIONAL CONVEX NEURAL NETWORKS

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

Rakesh Achanta
June 2019

This dissertation is online at: http://purl.stanford.edu/dz563hn6234

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

Trevor Hastie, Primary Adviser

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

Jonathan Taylor

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

Robert Tibshirani

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Our work is an expression of our Joy.

SADHGURU

Abstract

This work falls at the intersection of three fields or research: Neural Networks, Boosting and Sparse Inverse Problems. Neural Networks, despite being very successful, are poorly understood. Convexification has been an important tool in analyzing a single hidden layered network in recent years. Boosting is one of the most popular tools available for non-linear regression and classification. While boosting is a very general idea, only one variety (greedy gradient boosting with trees) seems to completely dominate the field, leaving scope for expanding the applicability of boosting; and also the algorithms used for fitting it. We employ a model for convex neural network which has been previously studied, but our focus will be on actually fitting it. In doing so, we also see that we can better understand boosting. This leads us to greatly expand both the applicability and algorithms for boosting. Given that our formulation of the convex neural network is a sparse inverse problem, the Julia based framework we develop can be used to solve any sparse inverse problem in the various methods we describe with minimal code development. Finally, as a bonus, we see that the methods we present are not just pedagogical but they do have competitive performance on real datasets. They match the performance of not only Neural Networks of comparable complexity that are hard to train, but also that of state-of-the-art boosting technologies like XGBoost with trees.

Acknowledgments

My time at Stanford had two strands to it; on one hand, there was the amazing academic atmosphere with inspired scientists all around and on the other hand, I was harmonizing various mysterious illnesses all through my time here. The challenges were great, both academically and physically, and the accomplishments equally rewarding. I could not have gone through this rigorous journey without the generosity of hundreds of friends that went way out of their way to support me in various ways. Here I try to acknowledge their help.

I begin by thanking my advisor Prof. Trevor Hastie. Apart from being a world class statistician, he is also deeply spiritual and kind. He always accommodated my varying condition. It has been a honour to work with the group that also has Prof. Rob Tibshirani and Prof. Jonathan Taylor, also very gentle while being highly gifted. Prof. Jerry Friedman's work has been an inspiration throughout. There is more to learn from these great scientists, in person, than even what is presented in "Elements of Statistical Learning" — the classic text that is to us dearer than the Bible. In addition, I would like to thank Prof. Ryan Tibshirani, also for his inspiration and his valuable comments and encouragement at the early stages of this work. I thank Prof. Stefan Wager for chairing my orals committee. I thank the various professors at Stanford for teaching Statistics in an inspiring way, these include (but are not limited to): Professors Art Owen, Joe Romano, Kannan Soundararajan, Ronald Howard, Lester Mackey, Persi Diaconis, Andrea Montanari, Emmanuel Candés, Sourav Chatterjee, Chris Manning, Stephen Boyd, John Duchi, etc. Special thanks to Prof. Robert Plummer, Prof. Vadim Khayms and Prof. David Dill for giving me the recommendation letters necessary for admission into the PhD program. Noah Simon encouraged me to apply for the PhD and also introduced me to the word autoimmunity. Special thanks to Prof. Fred Porta for making my time here more poetic through his Sanskrit class.

Chronologically I thank Julia Fukuyama, Alex Svistinov, Gourab Mukherjee, etc. for their guidance during my clueless MS time here. During my PhD, Kris Sankaran and Snighda Panigrahi helped me get through the very hard first year and the subsequent qualifying exam. The rest of the cohort, Jessy Hwang, Joey Arthur, Subhabrata Sen, Zhou Fan, Jiyao Kou and Nan Bi made it a lot of fun. I got really fortunate with taking an year off to join the next cohort and making friends with the funsane group Jelena Markovic, Leying Guan, Keli Liu, Pragya Sur, Eugene Katsevich, Wenfei

Du and Paulo Orenstein. I need to specially thank Junyang Qian, my office-mate for all the feedback on all my crazy ideas that usually boiled down to a Linear Algebra factoid. Charles Zheng played a key role in shaping my research in its early days, and gave me an opportunity to collaborate; we are also fellow seekers on the road to Truth. Feng Ruan's insights into the hard sub-problem were very helpful. With this PhD, I join an amazing lineage including fellow Trevorers of my time here, Will Fithian, Qingyuan Zhao, Charles Zheng, Ya Le, Junyang Qian, Elena Tuzhilina and Zijun Gao. It was always a festival in the department hanging out with the likes of Bhaswar Bhattacharya, Joshua Loftus, Haben Michael, Cyrus DiCiccio, Kelvin Guu, Lorenzo Cappello, Stephen Bates, Claire Donnat, Rina Friedberg, Mona Azadkia, Qian Zhao, Suyash Gupta, Swarnadip Ghosh, etc. Special thanks to Nikolaos Ignatiadis for his insights into both the computational and mathematical aspects of my work; similarly to my officemates Joey Arthur and Michael Sklar for all the statistical and philosophical discussions; and to Susie Ementon for making sure we did all we needed to in order to graduate — while also being our unpaid therapist. Supporting her, supporting us, were Caroline Gates, Ellen Van Stone, etc.

On the personal side I would first thank my immediate family — parents, sister, brother-in-law — for their support. Then I would like to thank my whole extended family, all my thousand cousins, especially Navya & Srini Yelavarthi, Aravinda & Kishore Chilukuri, Lavanya & Veerendra Achanta, Praveen Duggirala, etc. I would like to thank my Telugu literature friends circle, including Dr. Ismail Penugonda and his friend Gurunath, Pramada & Suresh Kolichala, Savitri & Narayanaswamy Motaparthi, Rahimanuddin Shaik, etc.; my friends from college Sudheer & Shivani Kondapalli, Cibin George, Kundana & Karthik Bandi, Bharat & Brinda Medasani, Roshni & Shaji Kunjumohamed, etc.; friends from my masters time, Avinayan Senthilvelayuthan, Tarun Vir Singh, Noa Tadd, Alex Svistinov, Vivek Kaul, Elizabeth & Chris Young, etc.; friends from Isha yoga, Ankur Jain, Preethi Shrikhande, Kavita & Avinash Satwalekar, Sivabalan, Chandini Ammineni, Ramya Chidambaram, Hema & Vijayan Narayanaperumal, Vivianne Nantel, Michael Levine, Kiah Bosy, Chitra & John Balasubramanium, Narendra Pengalur, Bonnie Hale, Shuba Somas, Sridevi & Venu Koneru, Reema Walia, Jay Krishnan, etc.; friends from Stanford PhD Kris Sankaran, Arun Prasad, Somik Raha, Sanjeev Sateesh, Matz Haugen, Sundari & Matthew Tappert, Derek Mendez, Prashanth Poddar, Charles Zheng, etc.; friends from Ramakrishna Mission's Swami Vedananda's class, Sandhya & Raghu Arur, Jean Yao, Aravindakshan Babu, Stephen Racunas, Haritha Eachampati, Ravi Kannan, Peter Diao, Fanya Becks, Patrick Horn, etc. Special thanks to the wonderful ladies of Spirit House, which was my spiritual home and refuge, Nirmala Johnson, Lorna Boich, Tara DeNuccio, Jordan Funk, Lydia Hwang, Siddhi Ellinghoven, Heather Bruchs, Brenda Herrington, Karen Gunn, etc. Extra special thanks to Beth and Sam Wilson for providing me housing for the last one year of my PhD! The final cherry on the top of the cake was my friendship with Amma's group in Santa Cruz led by Alison & John Richards. My healers in the alternate path of harmonizing the spirit and the body, Laura Fraser, Pratichi Mathur, Bhuvaneshvari Kaviraj, Robert Rowen, Toni & Silvano Senn, Jedd Krivcher, Jean Yun, Susan Cheng, Melvin Friedman, Nina & Niha Bhoot, Deepak Bhandari, Suhas Kshirsagar, etc. have taught me more about life itself than Stanford did. The immediate friendship of complete strangers like Elizabeth Maxim, Peter Sullivan, Jeanene Tremoulet and other friends of the Electro-sensitive group in an antagonistic world reassured hope. Tim Hastie gave me the courage to follow my heart post Stanford.

Finally I would like to acknowledge mononucleosis, tuberculosis, gerd, peptic ulcers, lupus, sun sensitivity, fibromyalgia, arthritis, chronic fatigue, lead poisoning, persistent gory nightmares, hot flashes, abdominal pain, pelvic floor dysfunction, root canal toxicity, insomnia, whiplash, hip misalignment, scoliosis, salt gluten and other food sensitivities, migraines, eye pain, electro-sensitivity and the resulting mental stress and loneliness for all the valuable lessons in *willpower* and *empathy*.

Contents

Chapter 1

Introduction

Feature Engineering is an integral part of modern data science, not only do data scientists spend significant amount of time trying to find good features, but also the most successful modern Machine Learning algorithms are those that are good at extracting features from the data. Prime examples of such algorithms are deep neural networks, where specialized architectures (like convolution) are combined with depth to get useful features from specific types of data (like images). Another very popular supervised learning algorithm is Boosting. While it is not obvious at first sight that Boosting is looking for good features, we can still think of the process of looking for 'weak-learners', as looking for features. For example a 'stub' or a tree with low complexity can be considered a feature; and boosting looks for such features in decreasing order of relevance.

On the other hand there is Feature Selection, which has been a major force in the field of Machine Learning over the last couple of decades. The Lasso[Tibshirani, 1996, Chen et al., 2001] and its variants offer a very comprehensive and well understood set of related methods based on ℓ_1 penalization to select informative features. The number of candidate features can be far greater than the number of data samples.

In our work we use the science of feature selection to do effective feature engineering automatically. Note however, that here we do not refer to the full extent of feature engineering that a data scientist with pertinent background knowledge in the field of application is capable of. We only refer to the ability of machine learning algorithms to select good functions of the data to build ensemble methods of.

Given n observations comprising the data

$$\{y_i, \mathbf{x}_i\}_1^n = \{y_i, x_{i1}, \cdots, x_{ip}\}_{i=1}^n,$$

where y_i is the "response" and $\mathbf{x}_i \in \mathbb{R}^p$ is the "predictor". We aim to estimate the "fit" $z_i \in \mathbb{R}$ from $\mathbf{x}_i$. Given our emphasis on ensemble methods for problem classes encompassing generalized 'non'-linear models, we will talk in terms of z_i, which might not belong to the same space as y_i. For example, in the case of non-linear logistic regression with the canonical link, $y_i \in \{0, 1\}$ is the response whereas $z_i \in \mathbb{R}$ is the log-odds ratio. The fit of z_i to y_i is measured by the loss $L(z_i; y_i)$. This loss encapsulates the model, like hinge loss for Support Vector Machines, negative log likelihood for a generalized (non)-linear model, etc. The loss can also be defined to encapsulate problems like matrix completion, image denoising, posterior likelihood, etc..

We model the ensemble 'fit' as

$$z = f(\mathbf{x}; \boldsymbol{\mu}) = \sum_{j=1}^{m} \beta_j \phi(\mathbf{x}; \omega_j), \tag{1.1}$$

where $\boldsymbol{\mu} = \{\omega_j, \beta_j\}_1^m$ are the model parameters, and ϕ is our choice of non-linearity or weak-learner. One popular choice for the weak-learner ϕ is the 'decision tree', with ω representing the collection of parameters required to describe the tree in full.

The data is a random sample from the distribution $\mathcal{P}(\mathbf{x}, y)$ and we want to minimize the expected loss or "risk",

$$R(\boldsymbol{\mu}) = \mathbb{E}_{\mathcal{P}} L\left(f(\mathbf{x}; \boldsymbol{\mu}); y\right). \tag{1.2}$$

The optimal parameter values are

$$\boldsymbol{\mu}^* = \operatorname*{argmin}_{\boldsymbol{\mu}} R(\boldsymbol{\mu}),$$

and an empirical estimate of the same is obtained by using the observed data instead of $\mathcal{P}$

$$\hat{\boldsymbol{\mu}} = \operatorname*{argmin}_{\boldsymbol{\mu}} \hat{R}(\boldsymbol{\mu}),$$

where

$$\begin{aligned} \hat{R}(\boldsymbol{\mu}) &= \tfrac{1}{n} \sum_{i=1}^{n} L\left(f(\mathbf{x}_i; \boldsymbol{\mu}); y_i\right) \\ &= \tfrac{1}{n} \sum_{i=1}^{n} L\left(\sum_{j=1}^{m} \beta_j \phi(\mathbf{x}_i; \omega_j); y_i\right). \end{aligned}$$

Our model so far is very flexible and without regularization $\hat{\boldsymbol{\mu}}$ could be a poor estimate of $\boldsymbol{\mu}^\star$, especially when n is small and m is big. To prevent over-fitting we regularize by imposing a constraint on each ω_j as $\omega_j \in \Omega$. Where Ω is an appropriately defined 'norm ball' or equivalent, depending on the choice of ϕ. We further regularize by imposing an ℓ_1 constraint on the coefficients $\boldsymbol{\beta} = \{\beta_1, \cdots, \beta_m\}$. This penalty can either be in lagrangian form (by adding $\lambda \|\boldsymbol{\beta}\|_1$ to the empirical loss) or in bound form (by restricting $\hat{\boldsymbol{\mu}}$ such that $\|\boldsymbol{\beta}\|_1 \leqslant C$). Finally we can use the number of terms in $\boldsymbol{\mu}$, in (1.1) as a regularization parameter.

Two popular methods that try to build ensemble non-linear models are Boosting and Neural Networks. Boosting is known to be the most popular off-the-shelf machine learning algorithm for unstructured data, while neural networks have gained immense popularity recently, especially given their ability to work with structured data like images and video. Here we will study what we call an 'Infinite Dimensional Convex Neural Network'. This model can be seen as a generalization of the boosting idea. Given that we are employing ℓ_1 penalty to encourage sparsity in infinite dimensions, our problem is also an instance of Sparse Inverse Problems. It can also be seen as a systematic approach to studying neural networks, starting with one with a single hidden layer. Our main aim is to explore connections between the various machine learning models and algorithms like the Lasso, LARS, Boosting, Stagewise methods, Kernel methods, Neural Networks, etc. Choices are made based on modeling preferences and fitting preferences. Most of the models come with explicit regularization (ℓ_1 or ℓ_2); while at other times, algorithmic regularization is incorporated while fitting. We view all the above from the lens of our problem — the "Infinite Dimensional Convex Neural Network". Our model aims to be a at the center of all the above models and methods; with modifications that can lead to each of them.

From the insights we obtain from the connections and similarities between various methods, we are better informed to build and engineer a function fitting framework that can be used to fit a wide range of models, with a wide range of algorithms, for various choices of data types, loss functions, non-linearities and regularizations. This allows us to experiment and see the similarities between apparently disparate models developed independently and further come up with one that has the good properties of all.

Chapter 2

Notation

We use bold letters to denote vectors like $\mathbf{y}, \boldsymbol{\beta}$, etc. Each individual element of $\mathbf{y}$ is given by y_i, etc. So $\mathbf{u}_1, \mathbf{u}_2$, etc. would all be vectors. Matrices are given in bold and are capitalized like $\mathbf{X}$, its i^{th} row is given as $\mathbf{x}_i$ and the j^{th} column as $\mathbf{X}_j$. Scalars like b, etc. are in normal face. x_{ij} denotes an element of $\mathbf{X}$. Random variables are denoted as X, Y, etc. regardless of dimension.

$\phi(\mathbf{x}; \omega)$ is the scalar value of the non-linear function ϕ, parameterized by ω applied to the datapoint $\mathbf{x}$. We overload the notation for ϕ to operate element-wise on both arguments. So $\phi(\mathbf{X}; \omega)$ is one *feature/predictor* n-vector obtained by applying, to each row of $\mathbf{X}$ individually, the mapping $\mathbf{x}_i \mapsto \phi(\mathbf{x}_i, \omega)$. $\phi(\mathbf{x}; \Omega)$ is the map $\phi(\mathbf{x}; \cdot) : \Omega \to \mathbb{R}$. It can also be considered a dense vector indexed continuously by $\omega \in \Omega$. Finally $\phi(\mathbf{X}; \Omega)$ can be *thought of* as the feature matrix with n rows and ∞ columns. Each row corresponds to the featurization of each datapoint $\mathbf{x}_i$. Each column is the training data feature defined by an $\omega \in \Omega$. Similarly we have $\phi(\mathbf{X}; \mathcal{A}) \in \mathbb{R}^{n \times |\mathcal{A}|}$ for any $\mathcal{A} \subset \Omega$. We use the boldface $\mathbf{\Phi}$ to also represent the object $\phi(\mathbf{X}; \Omega)$ when it is seen as a linear operator that takes a candidate measure and gives the fit corresponding to that measure.

$L(z_i; y_i)$ is the loss evaluated at one datapoint $(\mathbf{x}_i, y_i)$. The total loss on the training data is represented in bold-face as $\mathbf{L}(\mathbf{z}; \mathbf{y}) = \sum_i L(z_i; y_i)$. An unqualified n or p refer to the dimensions of the data matrix $\mathbf{X}_{n \times p}$. We usually index the datapoints with an i (as in $\mathbf{x}_i, y_i, z_i$, etc.) and the variables or features with j (as in $\mathbf{X}_j$ or ω_j).

Chapter 3

Background

Ensemble methods employing non-linear embeddings of the input data have a long history in Machine Learning. Here we present a few as background before we present our model. We leave out averaging methods like Random Forests and Bagging as they do not fit our narrative seamlessly given our focus on methods that fit sequentially rather than in parallel — i.e., the choice of parameters for the first weak-learner informs those of the second etc., which is not the case in Random Forests. We present Projection Pursuit Regression, Hinging Hyperplanes and One Hidden Layer Neural Network as ensemble methods of interest. Then we also present Trend Filtering, which fits a piecewise linear function in 1D. This can indeed be thought of as an ℓ_1 penalized ensemble method in 1D. Having good theoretical properties, it motivates us to look for generalizations into higher dimensions, which our model eventually does.

3.1 Projection Pursuit Regression

Projection Pursuit Regression(PPR) [Friedman and Tukey, 1974] models the function as

$$f(\mathbf{x}) = \sum_{j=1}^{m} \beta_j \phi_j(\mathbf{x} \cdot \mathbf{u}_j), \tag{3.1}$$

The key difference between this and the ensemble methods presented in (1.1) is that the non-linearity can be any function of a projection. For each term in the ensemble, a different function ϕ_j can be chosen to best minimize the empirical loss. There is no other regularization except the number of terms and the restriction that $\mathbf{u}_j$ is a unit-norm directional vector. PPR is designed for Mean-Squared Error Loss (MSE). It is fit in a forward step-wise fashion. At iteration k, we minimize the loss with respect to the current residuals. We initialize $\mathbf{u}_j$ randomly, and the function ϕ_j is a smoothener. The nonlinearity is approximated by a first order Taylor expansion at $\mathbf{u}_j$ and the

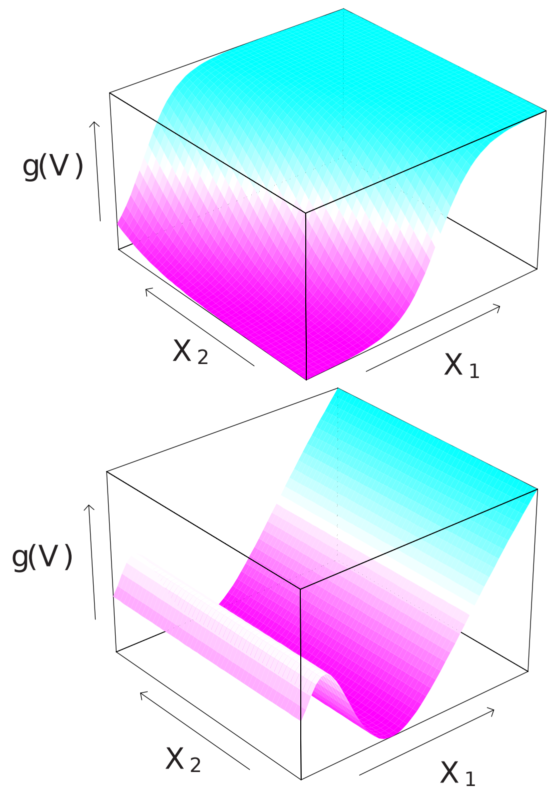

Figure 3.1: Perspective plots of two ridge functions used in PPR from [Friedman and Stuetzle, 1981].

resulting linear least squares problem is minimized to get an update for $\mathbf{u}_j$. Given this new $\mathbf{u}_j$, we smoothen the fit to get updated ϕ_j . This is repeated till convergence. In short it is a forward step-wise alternating Gauss-Newton algorithm in $\mathbf{u}_j$ and ϕ_j [Gill et al., 1981]. In later years the lasso has proved itself to be a better fitting algorithm than forward stepwise for the most part [Efron et al., 2004]; we will incorporate that wisdom to make PPR better.

3.2 Hinging Hyperplanes

Hinging Hyperplanes (HH) [Breiman, 1993], although independently developed, is very similar to PPR. Here the non-linearity ϕ is fixed to be a hinge function, which is the maximum (or minimum) of two linear functions of $\mathbf{x}$.

$$\phi(\mathbf{x}; \mathbf{u}, b, \mathbf{v}, c) = \pm \max(\mathbf{x} \cdot \mathbf{u} + b, \mathbf{x} \cdot \mathbf{v} + c) \tag{3.2}$$

On close observation, the fitting algorithm is the same as PPR. It is just the forward step-wise alternating Gauss-Newton method (specialized to hinge functions).

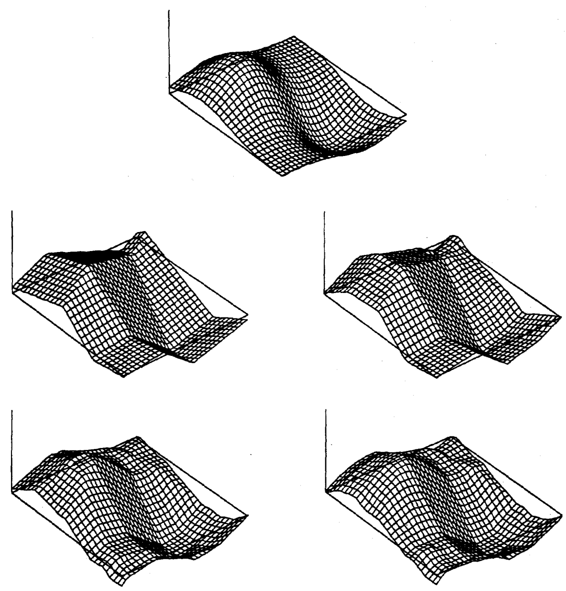

Figure 3.2: A series of fits using Hinging Hyperplanes. True function is at the top. Sourced from [Breiman, 1993].

3.3 One Hidden Layer Neural Network

A one hidden-layer neural network(NN) takes an input $\mathbf{x} \in \mathbb{R}^p$ and tries to predict $z \in \mathbb{R}$. Each node of the hidden layer represents a projection of $\mathbf{x}$ along $\mathbf{u}$ (and addition of a bias b) and application of a non-linearity ϕ. These nodes are combined (via a vector $\boldsymbol{\beta}$) to give the prediction z. The aim is to find the parameters $\boldsymbol{\mu} \equiv (\boldsymbol{\beta}, \{\mathbf{u}_j, b_j\})$ to

$$\min_{\boldsymbol{\mu}} \sum_{i=1}^{n} L\left(z_i; y_i\right), \tag{3.3}$$

where

$$z_i = \sum_{j=1}^{m} \beta_j \; \phi\left(\mathbf{u}_j \cdot \mathbf{x}_i + b_j\right). \tag{3.4}$$

A popular choice (and ours too) is the ReLU activation, which has played a key role in the successful comeback of of neural networks [Nair and Hinton, 2010].

$$\phi(t) = \mathrm{ReLU}(t) = (t)_+ = \max(0, t).$$

To solve this problem, we initialize $\boldsymbol{\beta}$ and $\{\mathbf{u}_j, b_j\}$ randomly and perform (Stochastic) Gradient Descent (SGD) with respect to them. The number of nodes m is fixed in advance.

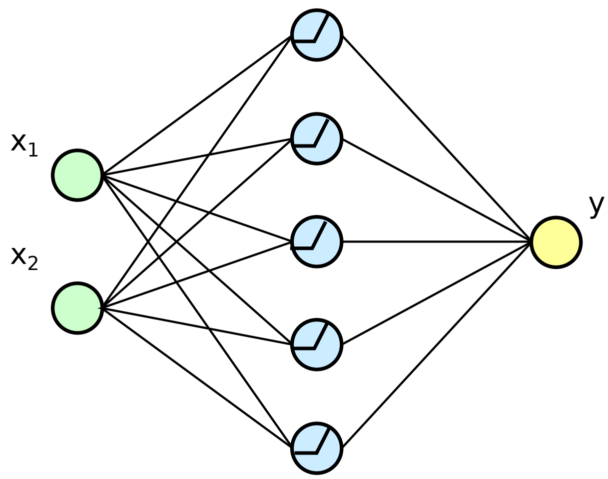

Figure 3.3: Neural Network in $\mathbb{R}^2$

This does not generally lead to a global minimum, but local minima are satisfactory in practice. It is also common to apply various penalties on $\mathbf{u}_j$ and $\boldsymbol{\beta}$. The regularized problem thus becomes:

$$\min_{\boldsymbol{\beta}, \|\mathbf{u}_j\|_2 \leqslant 1, b_j} \sum_{i=1}^{n} L\left(\sum_{j=1}^{m} \beta_j \ (\mathbf{u}_j \cdot \mathbf{x}_i + b_j)_+ ; y_i\right) + \lambda \|\boldsymbol{\beta}\|_1 . \tag{3.5}$$

Note that we can restrict the norm of $\mathbf{u}_j$ to be unity, since its magnitude can be pulled out of the ReLU into $\boldsymbol{\beta}$. Hence there is no reduction in model complexity. Once we restrict $\|\mathbf{u}_j\| \leqslant 1$, we can also restrict

$$|b_j| \leqslant \max_i |\mathbf{x}_i \cdot \mathbf{u}_j| \leqslant \max_i \|\mathbf{x}_i\| .$$

This is because any b outside this range does not introduce a non-linearity (at least as far as the training data are considered).

Thus a one hidden layer neural network can be considered an ensemble method of non-linear embeddings of the input data. However, atypical of ensemble methods, neural networks are not built sequentially (where one node is added after another); they are also not built in parallel like averaging methods are (where each weak-learner is learnt completely independently of the rest).

3.4 Trend-Filtering

We now present an important and interesting algorithm which has the same model as a ReLU activated Neural Network. Trend-Filtering(TF) [Kim et al., 2009], also a special case of Fused Lasso [Tibshirani et al., 2005], applies to $x \in \mathbb{R}$ with Mean Squared Error (MSE) loss. Given the data — ordered predictor $\{x_1, \cdots, x_n\} \subset \mathbb{R}$ and corresponding outcome $\{y_1, \cdots, y_n\} \subset \mathbb{R}$ — we try to fit

$\{z_1, \cdots, z_n\}$ as

$$\min_{\mathbf{z}} \sum_{i=1}^{n} (y_i - z_i)^2 \quad \text{s.t.} \quad \sum_{i=1}^{n-1} \left| \frac{z_{i+1} - z_i}{x_{i+1} - x_i} \right| \leqslant C. \tag{3.6}$$

Notice that the penalty is sum of deviations from linearity of z_i (as a function of x_i). Hence in effect, we are trying to find a piece-wise linear function of x.

The above problem can be written in the corresponding synthesis form [Tibshirani, 2014, Elad et al., 2006] as

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} \left(y_i - \alpha_0 - \alpha_1 x_i - \sum_{j=1}^{n} \beta_j (x_i - x_j)_+ \right)^2 \quad \text{s.t.} \quad \|\boldsymbol{\beta}\|_1 \leqslant C. \tag{3.7}$$

Or equivalently,

$$\min_{\boldsymbol{\beta}} \|\mathbf{r} - \mathbf{B}\boldsymbol{\beta}\|_2^2 \quad \text{s.t.} \quad \|\boldsymbol{\beta}\|_1 \leqslant C. \tag{3.8}$$

Where $\mathbf{r}$ is the residuals of regressing $\mathbf{x}$ out of $\mathbf{y}$ and $\mathbf{B}_{n\times n}$ is the linear spline basis matrix with knots at datapoints $\{x_i\}$. This is a simple Lasso problem, whose solution has been shown to be asymptotically minimax as $n \uparrow \infty$ for a given C, for equally spaced x_i on a fixed interval [Tibshirani, 2014].

Now we draw an interesting connection between the above two models, TF and NN, under MSE loss. It is directly apparent that they both fit piece-wise linear functions. We can simulate TF (3.7) using the neural network (3.5) by taking $p = 1$, $N = n$, $\mathbf{u}_j = 1$ and $b_j = x_j$. TF theory suggests that $b_j = x_j$ is optimal. Moreover, TF is a convex problem and can be solved efficiently; while a general NN is not. It is also equivalent to "locally adaptive regression splines" [Mammen et al., 1997].

In the one dimensional case, when moving from (3.5) to (3.7), given that $\mathbf{u}_j \in \{-1, 1\}$, we could pick it to be $+1$ without loss of model expressibility. But in higher dimensions, the set $\{\|\mathbf{u}_j\|_2 = 1\}$ becomes much more complex and no clear choice of directions arises. The same applies to choice of b. In this work we formulate an extension of TF to higher dimensions and try to solve it efficiently for a 'path' of regularization parameters.

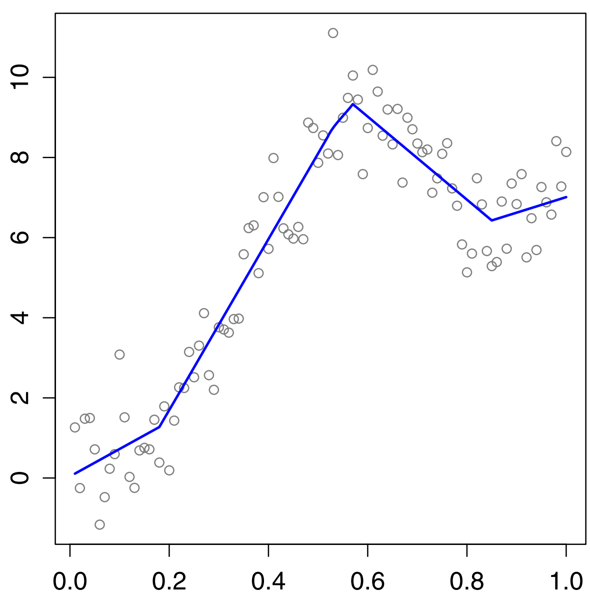

Figure 3.4: Trend filtering works in 1D and fits a piece-wise linear function with knots at the datapoints, to minimize MSE loss.

Chapter 4

Formulation of Our Problem

We now present the infinite dimensional version of the neural network from chapter 3. We got the idea to formulate the feature engineering problem as an infinite dimensional Lasso problem along the lines of the ℓ_1 regularized sparse inverse problems of [Boyd et al., 2015], but it has already been presented first in [Bengio et al., 2006], and studied in detail in [Bach, 2014]. [Rosset et al., 2007] also solve an infinite dimensional lasso via the LARS, although they apply their algorithm to a much simpler one dimensional application that boils down to Trend-Filtering.

4.1 The Model

Recalling our model from (1.1),

$$z = \sum_{j=1}^{m} \beta_j \phi(\mathbf{x}; \omega_j),$$

we can think of $\boldsymbol{\beta}$ as a "general mass function" $j \mapsto \beta_j$ on the set $[m]$.

$$\boldsymbol{\beta} : \{1, \cdots, m\} \to \mathbb{R}$$

A general mass function is like a probability mass function, but without the positivity and sum-to-unity constraints. Then z is the 'integral' (or in probability theoretic terms 'expectation') of the function $j \mapsto \phi(\mathbf{x}; \omega_j)$ with respect to the 'measure' $\boldsymbol{\beta}$, which itself is defined with respect to the counting measure on $[m]$. Our aim is to reformulate z as an integral with respect to a general measure on more complicated sets than the simple $[m]$. The measure could have a mass or density function with respect to an underlying measure.

As a second step in our thought experiment, given our ensemble model above, we can think of its parameters $\boldsymbol{\mu} = \{\omega_j, \beta_j\}_1^m$ as an atomic measure with atom locations ω_j and corresponding mass/weights β_j.

$$\boldsymbol{\mu}(\cdot) = \sum_{j=1}^{m} \beta_j \delta_{\omega_j}(\cdot)$$

Here δ_ω represents the measure with a single atom at ω.

$$\delta_\omega(A) = \mathbb{I}\left(\omega \in A\right) \quad \text{for any} \quad A \subset \Omega. \tag{4.1}$$

Now we can write z as

$$z = f(x; \boldsymbol{\mu}) = \int_\Omega \phi(\mathbf{x}; \omega) d\boldsymbol{\mu}(\omega). \tag{4.2}$$

This extends the idea beyond atomic measures, since here (Ω, Σ) can be any measurable space. If Ω is finite or countable, we take Σ to be 2^Ω; else for $\Omega \subset \mathbb{R}^d$, we take it to be the Borel sigma algebra. $\boldsymbol{\mu}$ is now any measure on this space.

Given our fit z, the risk we are interested in minimizing remains the same as (1.2).

$$R(\boldsymbol{\mu}) = \mathbb{E}_{\mathbf{x},y} L\left(f(\mathbf{x}; \boldsymbol{\mu}); y\right). \tag{4.3}$$

4.2 Regularization

We now generalize the ideas of regularization presented for $\hat{\boldsymbol{\mu}}$ of (1) in chapter 1 to our new definition of $\boldsymbol{\mu}$ as a measure. The constraint on the parameters ω_j are encapsulated in Ω — the support of $\boldsymbol{\mu}$. For e.g., if we are using ReLU for ϕ, then the parameters are $\omega = (\mathbf{u}, b)$. A restriction on the size of ω can be captured in Ω as

$$\Omega := \left\{\omega = (\mathbf{u}, b) \middle| \mathbf{u} \in \mathbb{R}^p, \|\mathbf{u}\|_2 \leqslant 1, |b| \leqslant 1\right\}.$$

The equivalent of ℓ_1 constraint on the coefficient part of $\boldsymbol{\mu}$, that is $\boldsymbol{\beta}$, is now

$$\|\boldsymbol{\mu}\|_1 := |\boldsymbol{\mu}|(\Omega) = \int_\Omega d|\boldsymbol{\mu}|(\omega). \tag{4.4}$$

We just integrate the absolute value of the measure over the constraint set Ω. Also throughout this work we fix $\boldsymbol{\mu}$ to have no support outside Ω. [1]

$$|\boldsymbol{\mu}|\left(\Omega^c\right) = 0.$$

Since $\boldsymbol{\mu}$ is not restricted to be atomic, a restriction on m, the number of atoms, does not apply. We fix the set Ω a priori and our goal is to find a $\boldsymbol{\mu}$ that minimizes the risk (4.3). Overloading

[1] We briefly violate this restriction in section 10.2 where we propose the use of measures that are a convolution of a kernel with atomic measures.

notation, we use $\phi(\mathbf{X};\omega)$ to mean row-wise application of ϕ on $\mathbf{X}$, giving us an n-vector called a *predictor*. Similarly $\phi(\mathbf{X};\Omega)$ is the set of all *predictors*, where each predictor is indexed by an $\omega \in \Omega$. We can represent the fit as

$$\mathbf{z} = \int_{\Omega} \phi(\mathbf{X};\omega)\ d\boldsymbol{\mu}(\omega).$$

This being a linear functional of $\boldsymbol{\mu}$, we denote it as

$$\mathbf{z} = f(\mathbf{X};\boldsymbol{\mu}) = \Phi\boldsymbol{\mu} = \int_{\Omega} \phi(\mathbf{X};\omega)\ d\boldsymbol{\mu}(\omega). \tag{4.5}$$

Our aim is to minimize the empirical loss with added regularization, giving us our main problem:

$$\min_{\boldsymbol{\mu}} \mathbf{L}\left(\Phi\boldsymbol{\mu};\mathbf{y}\right) \quad \text{s.t.} \quad \|\boldsymbol{\mu}\|_1 \leqslant C. \tag{4.6}$$

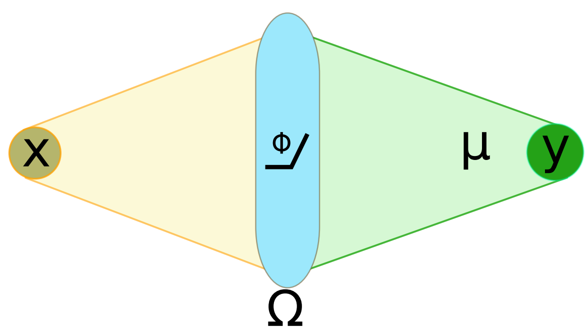

Figure 4.1: ∞D Convex Neural Network. Here the hidden layer is no longer a discrete set of nodes indexed by $\{1, 2, \cdots, m\}$, but it is a dense set of continuous 'nodes' indexed by $\omega \in \Omega$.

4.3 Examples

We now present a few examples of famous Lasso like problems that are but special cases of our very generic model based on measures.

4.3.1 Traditional Lasso

Here there is no featurization of the predictors, they are used as-is. Hence there are only p predictors. Ω, therefore, is $\{1, \cdots, p\}$ and ω is restricted to the p discrete values. ϕ is just the column picking function. For a given $\omega \equiv j$, $\phi(\mathbf{x}_i, j) = \mathbf{x}_{ij}$ and $\phi(\mathbf{X}, j) = \mathbf{X}_j$. $\boldsymbol{\mu} \equiv \boldsymbol{\beta}$, the p vector with some zeros possibly. $\mathbf{z} = \mathbf{X}\boldsymbol{\beta}$. If $\mathbf{L}$ is MSE, we recover traditional lasso in bound form as

$$\min_{\boldsymbol{\beta}} \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 \quad \text{s.t.} \quad \|\boldsymbol{\beta}\|_1 \leqslant C. \tag{4.7}$$

4.3.2 Gaussian Kernel Ridge Regression

Given one dimensional data

$$\mathbf{X}_{n\times 1} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},$$

and the gaussian kernel

$$\begin{aligned} k(x, x') &= \exp\left(-\tfrac{1}{2}[x - x']^2\right) &(4.8)\\ &= \exp\left(-\tfrac{1}{2}x^2\right)\exp\left(-\tfrac{1}{2}x'^2\right)\exp\left(xx'\right), &(4.9) \end{aligned}$$

we can obtain the predictor function ϕ from the Taylor expansion of the gaussian function as

$$\phi(x; \Omega) = e^{-x^2/2}\left[\frac{x^0}{\sqrt{0!}}, \frac{x^1}{\sqrt{1!}}, \frac{x^2}{\sqrt{2!}}, \frac{x^3}{\sqrt{3!}}, \cdots\right].$$

(similarly for $\phi(x'; \Omega)$), such that

$$k(x, x') = \langle \phi(x; \Omega), \phi(x'; \Omega) \rangle.$$

Hence we have a countably infinite number of predictors making $\Omega = \mathbb{N}$. For a given $\omega = j \in \mathbb{N}$,

$$\phi(x; j) = e^{-x^2/2}\ \frac{x^j}{\sqrt{j!}}.$$

This means that the linear functional Φ can be represented as an $n \times \aleph_0$ 'matrix' as

$$\Phi = \begin{bmatrix} e^{-x_1^2/2} & & & \\ & e^{-x_2^2/2} & & \\ & & \ddots & \\ & & & e^{-x_n^2/2} \end{bmatrix} \begin{bmatrix} 1 & x_1 & x_1^2/\sqrt{2} & x_1^3/\sqrt{3!} & \cdots \\ 1 & x_2 & x_2^2/\sqrt{2} & x_2^3/\sqrt{3!} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2/\sqrt{2} & x_n^3/\sqrt{3!} & \cdots \end{bmatrix}$$

Similarly, $\boldsymbol{\mu}$ is an infinite vector of coefficients. Here we use ℓ_2 penalty on $\boldsymbol{\mu}$ instead of the ℓ_1 in our lasso problem. The ridge regression problem is now

$$\min_{\boldsymbol{\mu}} \|\Phi\boldsymbol{\mu} - \mathbf{y}\|_2^2 \quad \text{s.t.} \quad \|\boldsymbol{\mu}\|_2 \leqslant C. \tag{4.10}$$

Of course this is not directly solved as we can not store an infinitely long matrix Φ or vector $\boldsymbol{\mu}$ in memory, neither can we resort to sparse representations. We instead use the kernel trick relying on the Caratheodary theorem (discussed in 4.5). This example demonstrates the breadth of our model.

4.4 Choices of ϕ, Ω

A choice of the non-linearity ϕ parameterized by ω also informs the choice of Ω. In this section we consider them jointly. Although the methods presented here can apply to any choice of ϕ (like trees, hinging hyperplanes, etc.), our focus in on the ReLU and its relatives, called positively homogeneous activation functions of integer degree [Bach, 2014].

$$\phi_\alpha(\mathbf{x}; \mathbf{u}, b) = [(\mathbf{x} \cdot \mathbf{u} + b)_+]^\alpha$$

$\alpha = 1$ corresponds to the Rectified Linear Unit (ReLU) and $\alpha = 2$ to the Rectified Quadratic Unit (ReQU). We choose these for several reasons. One, they can approach any measurable function [Leshno et al., 1993, Bach, 2014]. Two, given their homogeneity, i.e.,

$$\begin{aligned}
\phi_\alpha(\mathbf{x}; a\mathbf{u}, ab) &= a^\alpha \; \phi(\mathbf{x}; \mathbf{u}, b) \\
\texttt{relu}(\mathbf{x}; a\mathbf{u}, ab) &= a \; \texttt{relu}(\mathbf{x}; \mathbf{u}, b) \\
\texttt{requ}(\mathbf{x}; a\mathbf{u}, ab) &= a^2 \texttt{requ}(\mathbf{x}; \mathbf{u}, b),
\end{aligned}$$

they are invariant to change of scale of both the data $\mathbf{X}$ and the measure $\boldsymbol{\mu}$. This eliminates the necessity to impose a tuning parameter on the size of Ω, and allows us to simply restrict (an appropriately defined) norm of $\mathbf{u}$ to unity. Finally, the ReLU has had considerable empirical success and is the go-to activation function for neural networks today.

Once we pick ReLU to be ϕ, we have its parameters $\omega \equiv (\mathbf{u}, b)$. As mentioned in the previous section, we need to restrict the magnitude of $\mathbf{u}, b$ for a restriction on the norm of $\boldsymbol{\mu}$ to be meaningful. Given that we are looking for meaningful directions to project the input on to, a natural choice for Ω is

$$\Omega_2 := \big\{(\mathbf{u}, b)\big|\, \|\mathbf{u}\|_2 \leqslant 1, |b| \leqslant 1\big\}. \tag{4.11}$$

Another natural choice is using the one-norm instead of the two-norm. This might help us pick sparse projections (a *bet on sparsity*)

$$\Omega_1 := \big\{(\mathbf{u}, b)\big|\, \|\mathbf{u}\|_1 \leqslant 1, |b| \leqslant 1\big\}. \tag{4.12}$$

Alternatively, given that $\phi(\mathbf{X}; \omega)$ is an individual predictor, it is analogous to X_j in the traditional Lasso setting. Hence to make the coefficients of each basis comparable, we might want to normalize

$\phi(\mathbf{X};\omega)$. This gives us:

$$\Omega_{\mathbf{X}} := \Big\{\omega \;\Big|\; \|\phi(\mathbf{X};\omega)\|_2 \leqslant 1\Big\}, \tag{4.13}$$

which in the case of ReLU becomes:

$$\Omega_{\mathbf{X}} := \Big\{(\mathbf{u}, b) \;\Big|\; \|(\mathbf{X}\mathbf{u}+b)_+\|_2 \leqslant 1\Big\}. \tag{4.14}$$

With this data-dependent choice of Ω, we are trying to keep the $\phi(\mathbf{X};\omega)$ comparable, giving them an equal chance to enter the model. To get a better idea of how this set looks like, let us ignore the bias and look at

$$\Big\{\mathbf{u} \;\Big|\; \|(\mathbf{X}\mathbf{u})_+\|_2^2 \leqslant n\Big\}.$$

If the rows of $\mathbf{X}$ are iid from a symmetric mean-centered distribution, as $n \uparrow \infty$ this set tends to the ellipse

$$\Big\{\mathbf{u} \;\Big|\; \tfrac{1}{n}\|\mathbf{X}\mathbf{u}\|_2^2 \leqslant 2\Big\} \to \big\{\mathbf{u} \;\big|\; \mathbf{u}^T\Sigma_{\mathbf{x}}\mathbf{u} \leqslant 2\big\}.$$

Thus, for high n, $\Omega_{\mathbf{X}}$ restricts $\mathbf{u}$ in the *quadratic norm* induced by $\Sigma_{\mathbf{x}} = Cov(X)$; much like Ω_1 and Ω_2 in their respective norms. However, we might have to be careful about the conditioning of the above covariance matrix in the interest of stability of chosen directions $\{\mathbf{u}_i\}$. It can be shown that the set $\Omega_{\mathbf{X}}$ is convex much like Ω_1 and Ω_2 are.

4.4.1 Closure under Negation

In most cases there is no sign constraint on the weak-learner. For example if all the values in the terminal nodes of a regression tree are reversed, it would still be a tree. But this is not always the case. For example the ReLU is always positive. This does not reduce the expressibility of our model in any way since the measure itself is not sign restricted to be positive. In toto, we need either the measure or the activation to be free to carry any sign. Sometimes it is much easier to deal with positive measures. Especially with the ℓ_1 constraint on its size as in (4.4), a positive measure is just a probability measure up to a scale factor.

Assumption 1 (Negation Closed)**.** An activation, constraint-set pair (ϕ, Ω) is said to be negation closed if

$$\forall\ \omega \in \Omega,\ \exists\ \bar{\omega} \in \Omega \quad \text{s.t.} \quad \phi(\mathbf{X};\omega) = -\phi(\mathbf{X};\bar{\omega}).$$

Sometimes this condition is naturally satisfied, for example in the case of trees. In the case of ReLUs where it does not intrinsically hold, we could redefine ω and Ω to make it hold. For $\phi(\mathbf{X};\omega) = (\mathbf{X}\mathbf{u}+b)_+$, we have $\omega = (\mathbf{u}, b)$; instead, we can define $\omega = (s, \mathbf{u}, b)$, where $s \in \{-1, 1\}$ is the sign of the activation. And the activation itself is defined as

$$\phi(\mathbf{X};\omega) = \phi(\mathbf{X};(s, \mathbf{u}, b)) = s(\mathbf{X}\mathbf{u}+b)_+. \tag{4.15}$$

Thus negation closure is satisfied as

$$\phi(\mathbf{X};(+1,\mathbf{u},b)) = -\phi(\mathbf{X};(-1,\mathbf{u},b)). \tag{4.16}$$

When the Negation Closed Assumption 1 holds, we can reformulate our problem as

$$\min_{\boldsymbol{\mu}} \mathbf{L}\left(\Phi\boldsymbol{\mu};\mathbf{y}\right) \quad \text{s.t.} \quad \boldsymbol{\mu}(\Omega) \leqslant C \quad \text{and} \quad \boldsymbol{\mu} \geqslant 0, \tag{4.17}$$

where

$$\boldsymbol{\mu}(\Omega) = \int_{\Omega} d\boldsymbol{\mu}(\omega).$$

Again, $\boldsymbol{\mu}(\Omega^c) = 0$. This is the non-negative infinite dimensional lasso. Although this formulation does not change the problem in anyway, we sometimes use it when it makes the exposition of an algorithm simpler (like in the case of Stagewise Algorithm 3).

4.5 Theory

We now discuss the theoretical properties of our infinite dimensional lasso problem as defined by (4.6) focusing on the existence of a finite dimensional solution. Most of the work presented here is a summary of properties from our three main references [Rosset et al., 2007, Bach, 2014, Boyd et al., 2015], adapted to directly apply to our problem (4.6).

Theorem 1. If Ω is compact and ϕ is continuous, then there exists a finite dimensional solution to (4.6), with at most $n+1$ atoms.

Proof. This theorem is proved via the following two propositions. □

Proposition 1. There exists a solution to (4.6), if the set

$$\phi(\mathbf{X};\Omega) := \{\phi(\mathbf{X},\omega) \mid \omega \in \Omega\} \subset \mathbb{R}^n$$

is compact. A sufficient condition for this to hold is that ϕ is continuous and Ω is compact, since the image of a compact set is compact when the mapping is continuous.

The proof can be found in [Rosset et al., 2007]. We note that the ReLU being continuous and Ω_1, Ω_2 and $\Omega_{\mathbf{X}}$ being compact means that our problem of interest has a solution.

Proposition 2. If there exists a solution to (4.6), then there exists one with at most $n+1$ atoms.

Proof. Finite or countable Ω:

When Ω is finite or countable, the underlying measure is the counting measure and any candidate $\boldsymbol{\mu}$ has a mass function. In this case, we just apply the Caratheodory's theorem for convex hulls

[Rockafellar, 1997], hence any optimal solution given by

$$\mathbf{z}^* = \Phi\boldsymbol{\mu}^* = \sum_{j=1}^{|\Omega|} \beta_j \phi(\mathbf{X}; \omega_j)$$

should be expressible as a *convex* combination of at most $n+1$ vectors from the set

$$\left\{\beta_j \phi(\mathbf{X}; \omega_j)\right\}_{j=1}^{|\Omega|}$$

Now we can construct a new measure $\hat{\boldsymbol{\mu}}$ containing the same convex combination of (β_j, ω_j) pairs from that subset of $n+1$ atoms. Since the combination is convex, we have $\|\hat{\boldsymbol{\mu}}\|_1 \leqslant \|\boldsymbol{\mu}^*\|_1$. The loss with $\hat{\boldsymbol{\mu}}$ is the same and the penalty bound is satisfied, making it an optimal solution (with no more than $n+1$ atoms).

Uncountable Ω*:*

The proof for this more complicated case can be found in [Rosset et al., 2007]. □

The above proposition provides a preliminary justification for looking for solutions that are atomic measures. Additionally, we can not store a general measure supported on a general Ω on a computer.

4.6 Relation to Kernels

Since the empirical loss we are trying to minimize in the problem depends on the n observed data samples, we have a representer theorem stating that the loss depends only on the function class evaluated at the observed values. By Caratheodory's theorem for convex cones [Rockafellar, 1997], the fitted values $\mathbf{z}$ may be decomposed into at most n basis functions $\phi(\cdot, \omega_i)$. That is $\boldsymbol{\mu}$ is comprised of at most n atoms. This is the basis of the kernel trick where an infinite dimensional problem is reduced to an n dimensional one. From the infinite basis functions, n kernel functions are obtained (by evaluation of the kernel function at each $\mathbf{x}_i$). This is possible for kernel methods in general because of the ℓ_2 penalty on the function complexity. (See the Gaussian Kernel Ridge Regression example above.) In general for ℓ_1 penalty on $\boldsymbol{\mu}$, the problem does not boil down to an n dimensional one in *known* basis functions. (Trend Filtering is a noteworthy exception.)

The ℓ_2 version of our problem (4.6) is

$$\min_{\boldsymbol{\mu}} \mathbf{L}\left(\Phi\boldsymbol{\mu}; \mathbf{y}\right) \quad \text{s.t.} \quad \|\boldsymbol{\mu}\|_2 \leqslant C, \tag{4.18}$$

where again,

$$\Phi\boldsymbol{\mu} = \int_\Omega \phi(\mathbf{X};\omega)d\boldsymbol{\mu}(\omega)$$
$$\|\boldsymbol{\mu}\|_2^2 = \int_\Omega \mu^2(\omega)d\Lambda(\omega).$$

Here μ is the density function of $\boldsymbol{\mu}$ with respect to the underlying lebesgue measure Λ over Ω. We thus have to restrict the function class here to those that can be parameterized by $\boldsymbol{\mu}$ with a square integrable density function. This can be a serious restriction, as, for e.g., a measure with a single atom does not belong to this restricted class. Square integrability (having a variance) is a stronger condition than integrability (having a mean) for finite measures.

We can solve this problem via the kernel function. For example, consider the ReLU activation with ℓ_2 constraint on the weights, $\|\mathbf{u}\|_2 \leqslant 1$, and without bias ($b = 0$), the kernel function is given by

$$\begin{aligned} k(\mathbf{x}, \mathbf{x}') &:= \mathbb{E}_\Lambda \left[(\mathbf{u}\cdot\mathbf{x})_+(\mathbf{u}\cdot\mathbf{x}')_+\right] \\ &= \frac{\|\mathbf{x}\|_2 \|\mathbf{x}'\|_2}{2\pi p}\left((\pi-\theta)\cos\theta + \sin\theta\right) \end{aligned}$$

where the expectation is w.r.to the uniform measure over the set $\|\mathbf{u}\|_2 \leqslant 1$ and θ is given by

$$\cos\theta = \frac{\langle \mathbf{x}, \mathbf{x}'\rangle}{\|\mathbf{x}\|_2 \|\mathbf{x}'\|_2}.$$

Kernel representations for this example and more general ones can be found in [Le Roux and Bengio, 2007, Cho and Saul, 2009], here we give but a simple example to demonstrate the connection.

Alternately one can use Random Sampling of predictors — equivalently $\mathbf{u}_i$ uniformly from the constraint set. Once the $\mathbf{u}_i$ are sampled and fixed, we only need to solve for the output weights — which is a ridge regression problem. Random Sampling has been shown to work for kernelizable problems ([Rahimi and Recht, 2008]), i.e. the error goes down as $1/\sqrt{m}$, where again m is the number of basis functions. No such results are available for the ℓ_1 penalized problem. However this approach (of lasso on randomly sampled basis directions) can be a good baseline to compare our more complicated ensemble methods presented below.

Chapter 5

First Order Solvers

Given our problem (4.6), we will now present a few first order solvers. We restate the problem we are tyring to solve:

$$\min_{\boldsymbol{\mu}} \mathbf{L}\left(\Phi\boldsymbol{\mu};\mathbf{y}\right) \quad \text{s.t.} \quad \|\boldsymbol{\mu}\|_1 \leqslant C, \tag{5.1}$$

where

$$\Phi\boldsymbol{\mu} = \int_\Omega \phi(\mathbf{X};\omega)d\boldsymbol{\mu}(\omega)$$

and

$$\|\boldsymbol{\mu}\|_1 = |\boldsymbol{\mu}|\left(\Omega\right) = \int_\Omega d|\boldsymbol{\mu}|(\omega).$$

It is immediately obvious that the solver would depend on the choice of the mutually dependent activation-constraint pair (ϕ, Ω). In the interest of efficient product development, we would focus on principles that underlie the various solvers, regardless of specific choices of ϕ and Ω. Also equally importantly, in the interest of pedagogy, we want to see what the basic building blocks of successful machine learning methods are. For example what is common between Gradient Boosting and Neural Networks? What are the subtle differences between Stagewise methods and Frank-Wolfe? We will identify the various sub-routines that these algorithms use so that we can reuse them.

We will start our discussion with Gradient Descent on Neural Networks. While this method of fitting differs significantly from the others we present, we place it here, as it uses only the gradient information and hence is a first order method. Then we present the algorithms that iteratively linearize and minimize the problem.

5.1 Neural Networks

Given Caratheodory's theorem, we know that a solution with at most $n+1$ neurons exits. So one can start with a Neural Network with $n+1$ nodes randomly initialized and use Gradient Descent.

The Ω constraint is implemented via projections of the $\mathbf{u}_i$ on to Ω. Similarly the norm constraint is implemented by projecting the co-efficient vector $\boldsymbol{\beta}$ onto the ℓ_1 ball of radius C. Hence we have a Projected Gradient Descent method. If our problem is expressed in the lagrangian form, we go with the usual Proximal Gradient Descent (PGD). Where the proximal step is the *soft-thresholding* $\text{prox}_{\|\cdot\|}$ operator.

Our measure can be written as

$$\boldsymbol{\mu} = \sum_{j=1}^{n} \beta_j \delta_{\omega_j} \tag{5.2}$$

and our problem becomes

$$\min_{\boldsymbol{\beta}, \{\omega_j\} \subset \Omega} \mathbf{L}\left(\sum_{j=1}^{n} \beta_j \phi(\mathbf{X}; \omega_j); \mathbf{y}\right) \quad \text{s.t.} \quad \|\boldsymbol{\beta}\|_1 \leqslant C \tag{5.3}$$

Although our original problem (5.1) is convex in the underlying variable, the measure $\boldsymbol{\mu}$, once we reformulate it in terms of the parameters of the atomic measure, $(\boldsymbol{\beta}, \{\omega_j\})$, it is no longer convex in this space. This is because the set of all measures with a fixed number of atoms (here $n+1$) is not convex. PGD can still be used, but it requires a lot of tuning, detailed in Table 5.2. This is the reason training Neural Networks has become notorious as a black art. The problems become worse as we add many more layers. Also, this formulation has too many hidden layer nodes. Although given that we are using an ℓ_1 penalty on $\boldsymbol{\beta}$, we could bet that there will be lot lesser nodes in the final solution (with a non-zero coefficient) and start with a much smaller hidden layer, as is done in practice.

Gradients

We now provide explicit formulae for gradient descent for randomly initialized Neural Network's parameters — $\boldsymbol{\beta}, \mathcal{A}$; where $\mathcal{A} := \{\omega_j\}_{j=1}^m \subset \Omega$ is the active set. First we have

$$\mathbf{z} = \phi(\mathbf{X}; \mathcal{A})\, \boldsymbol{\beta} = \sum_{j=1}^{m} \beta_j \phi(\mathbf{X}; \omega_j).$$

Here $\boldsymbol{\beta} \in \mathbb{R}^m$, for a fixed m. We are trying to minimize $\mathbf{L}\left(\mathbf{z}; \mathbf{y}\right)$. By the chain rule,

$$\nabla_{\boldsymbol{\beta}} \mathbf{L}\left(\mathbf{z}; \mathbf{y}\right) = \nabla_{\boldsymbol{\beta}} \mathbf{z}^T \nabla_{\mathbf{z}} \mathbf{L}\left(\mathbf{z}; \mathbf{y}\right). \tag{5.4}$$

Where the first term $\nabla_{\boldsymbol{\beta}} \mathbf{z} \in \mathbb{R}^{n \times m}$ is the jacobian of $\mathbf{z} \in \mathbb{R}^n$ w.r.to $\boldsymbol{\beta} \in \mathbb{R}^m$. $\nabla_{\mathbf{z}} \mathbf{L} \in \mathbb{R}^n$ is the gradient of the loss $\mathbf{L} \in \mathbb{R}$ w.r.to $\mathbf{z} \in \mathbb{R}^n$. We have

$$z_i = \boldsymbol{\beta} \cdot \phi(\mathbf{x}_i; \mathcal{A})$$

This gives us

$$(\nabla_{\boldsymbol{\beta}} \mathbf{z})_{ij} = \phi(\mathbf{x}_i; \omega_j)$$

Using our overloaded notation we can express this very succinctly as

$$\nabla_{\boldsymbol{\beta}} \mathbf{z} = \phi(\mathbf{X}; \mathcal{A}) \in \mathbb{R}^{n \times m}. \tag{5.5}$$

Thus we have (given our definition of the gradient $\mathbf{g} = \nabla_{\mathbf{z}} \mathbf{L}(\mathbf{z}; \mathbf{y})$)

$$\boxed{\nabla_{\boldsymbol{\beta}} \mathbf{L} = \phi(\mathbf{X}; \mathcal{A})^T \mathbf{g}} \tag{5.6}$$

Now let us get the gradient of $\mathbf{L}$ w.r.to ω_j. Using the chain-rule,

$$\nabla_{\omega_j} \mathbf{L}(\mathbf{z}; \mathbf{y}) = \nabla_{\omega_j} \mathbf{z}^T \nabla_{\mathbf{z}} \mathbf{L}(\mathbf{z}; \mathbf{y}).$$

Again,

$$\mathbf{z} = \sum_{j=1}^{m} \beta_j \phi(\mathbf{X}; \omega_j).$$

So

$$\nabla_{\omega_j} \mathbf{z} = \beta_j \nabla_{\omega} \phi(\mathbf{X}; \omega_j).$$

This gives us the final gradient for each ω_j:

$$\boxed{\nabla_{\omega_j} \mathbf{L} = \beta_j \nabla_{\omega} \phi(\mathbf{X}; \omega_j)^T \mathbf{g}} \tag{5.7}$$

Gradient for ReLU

We have $\omega = (\mathbf{u}, b)$ and $\phi(\mathbf{X}; \omega) = (\mathbf{X}\mathbf{u} + b)_+ \in \mathbb{R}^n$. This means we have two parts to $\nabla_{\omega} \mathbf{L}$, which are $\nabla_{\mathbf{u}} \mathbf{L}$ and $\nabla_b \mathbf{L}$.

$$\begin{aligned} \mathbf{J}^{(\mathbf{u})} &= \nabla_{\mathbf{u}} \phi(\mathbf{X}; (\mathbf{u}, b)) \\ &= \nabla_{\mathbf{u}} (\mathbf{X}\mathbf{u} + b)_+ \end{aligned}$$

The jacobian w.r.to $\mathbf{u}$ is $\mathbf{J}^{(\mathbf{u})} \in \mathbb{R}^{n \times p}$ can be evaluated one row at a time. Where each row corresponds to one data sample $\mathbf{x}_i$.

$$\mathbf{J}^{(\mathbf{u})}_{i\cdot} = \begin{cases} \mathbf{0} & \text{if} \quad \mathbf{x}_i \cdot \mathbf{u} + b < 0 \\ \mathbf{x}_i & \text{else} \end{cases}$$

Similarly

$$\mathbf{J}^{(b)} = \nabla_b(\mathbf{X}\mathbf{u}+b)_+ = \mathbb{I}\left(\mathbf{x}_i \cdot \mathbf{u} + b \geqslant 0\right).$$

When we put these two together (and define $\tilde{\mathbf{X}}$ as $\mathbf{X}$ augmented with a leading column of 1's and move b into $\mathbf{u}$), we have the jacobian of $\mathbf{z}$ w.r.to an ω of interest as:

$$\boxed{\nabla_\omega \phi(\mathbf{X}; \omega) = \mathbf{J}(\omega) = \mathbb{1}_\omega \tilde{\mathbf{X}}} \tag{5.8}$$

Where $\mathbb{1}_\omega \in \mathbb{R}^{n \times n}$ is a diagonal matrix with

$$(\mathbb{1}_\omega)_{ii} = \mathbb{I}\left(\mathbf{x}_i \cdot \mathbf{u} + b \geqslant 0\right).$$

Which just means that the jacobian is equal to $\tilde{\mathbf{X}}$ with the inactive rows zeroed out based on the ω of interest. This saves us a lot of computation; as the computation of jacobian becomes just picking the rows of $\mathbf{X}$ that the particular ReLU is active on.

$$\boxed{\nabla_{\omega_j} \mathbf{L} = \ \beta_j \tilde{\mathbf{X}}^T \mathbb{1}_{\omega_j} \mathbf{g}} \tag{5.9}$$

Here $\beta_j \in \mathbb{R}, \tilde{\mathbf{X}} \in \mathbb{R}^{n \times p+1}$, $\mathbb{1}_{\omega_j} \in \mathbb{R}^{n \times n}$ and $\mathbf{g} \in \mathbb{R}^n$, meaning $\nabla_{\omega_j} \mathbf{L} \in \mathbb{R}^{p+1}$.

Algorithm 1 Neural Network

Input: $\mathbf{X}, \mathbf{y}$
Choices: $\phi, \Omega, \mathbf{L}, m, C$
Initialize: $\boldsymbol{\mu} = (\boldsymbol{\beta} \in \mathbb{R}^m, \{\omega_j\}_{j=1}^m)$ randomly
while not converged **do**
  $\mathbf{g} \leftarrow \nabla_{\mathbf{z}} \mathbf{L}$
  for $j \in \{1, 2, \cdots, m\}$ **do**
    $\omega_j \leftarrow \omega_j - \alpha_\omega \nabla_{\omega_j} \mathbf{L}$ ▷ Using (5.7)
  $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \alpha_\beta \nabla_{\boldsymbol{\beta}} \mathbf{L}$ ▷ Using (5.6)
  Project $\boldsymbol{\beta}$ onto the ℓ_1 ball of radius C
Return: $\boldsymbol{\mu} \equiv (\boldsymbol{\beta}, \{\omega_j\})$

Subroutines

From Algorithm 1 for training a neural network via gradient descent on the parameters $\boldsymbol{\mu}$, we see that there are a few subroutines that we might be able to reuse. Here we identify them and name them so that we can recognize them when we see them again in other algorithms.

1. **`gradient`**

   We want to calculate the gradient $\nabla_{\mathbf{z}} \mathbf{L}\left(\mathbf{z}; \mathbf{y}\right)$. We call this subroutine *gradient* or in boosting terms *negative generalized residual*. When the loss is MSE, this gradient is just the negative residual. The calculation of this subroutine is specialized for each loss; and the inputs for it are the constant $\mathbf{y}$ and the variable $\mathbf{z}$.

2. **`jacobian`**

   Then we have the jacobian of $\nabla_\omega \phi(\mathbf{X}; \omega)$. This subroutine is specialized for each activation ϕ, as defined for the ReLU by (5.8). It takes the constant $\mathbf{X}$ and the variable ω as inputs.

3. **`backprop`**ω

   We isolate the step $\omega_j \leftarrow \omega_j - \alpha_\omega \nabla_{\omega_j} \mathbf{L}$ from Algorithm 1 and give it a name. Although we are just using the chain rule here, we will call it by the broader technique *back propagation* to distinguish it from the various other gradient based updates we use.

5.2 Central Idea: Linearize, Minimize

In the following sections we present our main algorithms of interest to solve (5.1). Given that our constraint set $\|\boldsymbol{\mu}\|_1 \leqslant C$ is a polyhedron in infinite dimensions and is hard to visualize, we keep our methods simple. We linearize at the current estimate and minimize the linearization. We know that linear functions over the ℓ_1 ball are minimized at a vertex. That is the minimum point has all the mass on one coordinate.

Consider minimizing the function $f(\theta)$, subject to the constraint that $\|\theta\|_1 \leqslant 1$. Figure 5.1 shows the level curves of a sample $f(\theta)$ over the domain $\|\theta\|_1 \leqslant 1$ in 2D.

At our current estimate θ_0 we have the first order Madhava-Taylor approximation,

$$\hat{f}_0(\theta) = f(\theta_0) + \nabla f(\theta_0) \cdot (\theta - \theta_0) \tag{5.10}$$

We now try to minimize this

$$\begin{aligned}
\theta_1 &= \operatorname*{argmin}_{\|\theta\|_1 \leqslant 1} \hat{f}_0(\theta) \\
&= \operatorname*{argmin}_{\|\theta\|_1 \leqslant 1} \nabla f(\theta_0) \cdot \theta
\end{aligned}$$

It is obvious that to minimize the above inner product, we ought to put all our mass on the coordinate that has the maximum absolute value in $\nabla f(\theta_0)$. This gives us a cardinal direction along a coordinate

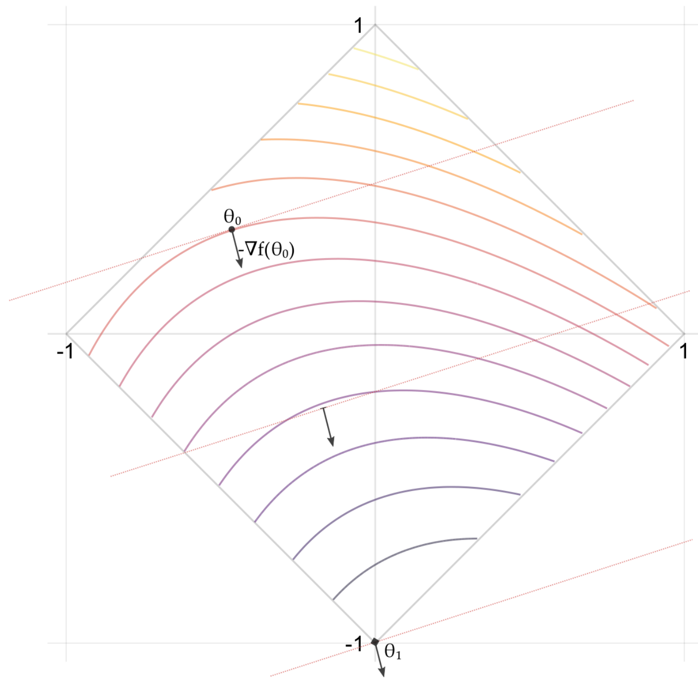

Figure 5.1: Demonstration of the linearize-minimize idea in 2D. Level curves are shown over the domain $\|\theta\|_1 \leqslant 1$. Level curves of the linearization at θ_0 are shown in dotted straight lines. The vertex θ_1 minimizes the linearization.

j^* as

$$\theta_1 = -s\ e_{j^*}$$

where,

$$j^* = \underset{j \in [p]}{\operatorname{argmax}} \left| \nabla_j f(\theta_0) \right|,$$

$$s = \operatorname{sign} \nabla_{j^*} f(\theta_0).$$

The sign is picked to minimize $\hat{f}_0(\theta)$; a negation of the sign would maximize $\hat{f}_0(\theta)$. In our toy example in 2D it is the vertex along the negative vertical axis. To update the iterate, θ_0 is augmented along j^*, by how much is left to individual algorithms that are based on this central idea.

In the infinite dimensional case of $\|\boldsymbol{\mu}\|_1 \leqslant C$. The idea of a vertex is less obvious. But it is easy to see the analogue of the cardinal direction $e_j[i] = \mathbb{I}\,(i = j)$ is the single atom measure $\delta_\omega(A) = \mathbb{I}\,(\omega \in A)$. We will revisit this idea in greater detail in the following sections.

5.3 Frank-Wolfe

Now let us consider a method that can be used to solve (5.1) for a fixed value of C. Frank-Wolfe (FW)[Jaggi, 2013] is a traditional method to minimize problems of the kind

$$\min_{\theta} f(\theta) \quad \text{s.t.} \quad \theta \in \mathcal{D}, \tag{5.11}$$

where f and $\mathcal{D}$ are convex. Our problem is also of this kind. FW iteratively linearizes and minimizes the problem. The minimum of each linearization is obtained on the boundary of $\mathcal{D}$, and if $\mathcal{D}$ is a polyhedron, the minimizer is a vertex of $\mathcal{D}$. As we are trying to minimize our problem (5.1) on an ℓ_1 ball, we will from now assume $\mathcal{D}$ to be a polyhedron. The vertex of the polyhedron that minimizes the linearization is added to an active set. And the next iterate for θ is a convex combination of these vertices in the active set.

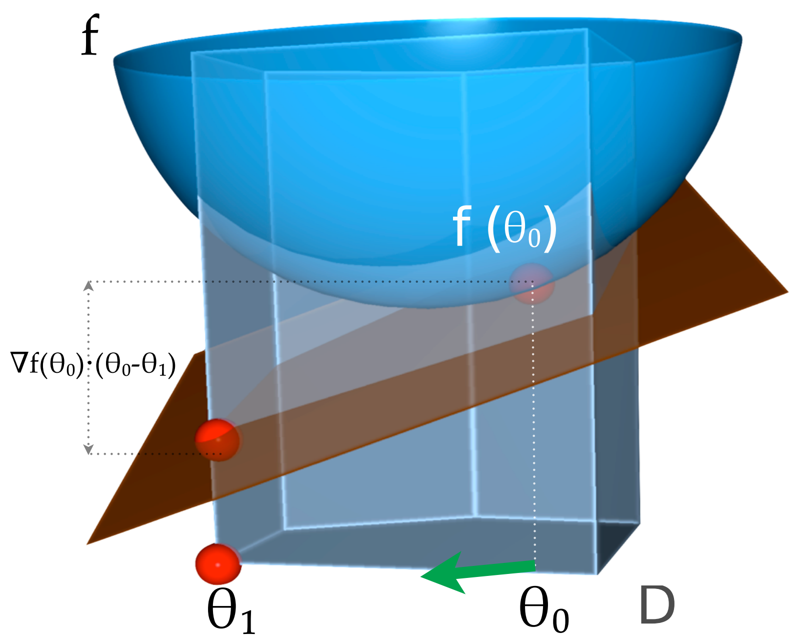

Figure 5.2: Demonstration of a Frank-Wolfe iteration from [Jaggi, 2013]. The brown plane represents the linearization at θ_0. θ_1 is the vertex of $\mathcal{D}$ where the linearization is minimized. The iterate is then moved *a bit* towards θ_1.

More elaborately, at the k^{th} iteration, given our current estimate for θ as $\theta^{(k)}$, FW does the following:

1. Linearize f around $\theta^{(k)}$ to get

$$f^{(k)}(\theta) = f(\theta^{(k)}) + \left\langle \nabla f(\theta^{(k)}), \theta - \theta^{(k)} \right\rangle \tag{5.12}$$

2. Minimize $f^{(k)}$ to get $s^{(k)}$ (which will be a vertex of $\mathcal{D}$)

$$s^{(k)} \in \operatorname*{argmin}_{\theta \in \mathcal{D}} \left\langle \nabla f(\theta^{(k)}), \theta \right\rangle \tag{5.13}$$

   and add it to the running active set of vertices $\mathcal{A}^{(k)} = \{s^{(1)}, \cdots, s^{(k)}\}$.

3. Minimize over the convex hull of $\mathcal{A}^{(k)}$ to get next iterate

$$\theta^{(k+1)} = \operatorname*{argmin}_{\theta \in \mathrm{conv}(\mathcal{A}^{(k)})} f(\theta) \tag{5.14}$$

Note that as we advance one iteration, the cardinality of the active set $\mathcal{A}$ increases by one. After k steps, we have the current solution,

$$\theta^{(k)} = \sum_{i=1}^{k} \beta_i^{(k)} s^{(i)},$$

where $\left\|\beta^{(k)}\right\|_1 = 1,\ \beta^{(k)} \succ 0$. Further more, when $\mathcal{D}$ is an ℓ_1 ball, each s_i is one-sparse making $\theta^{(k)}$ k-sparse. FW has a convergence rate of $O(1/k)$ for smooth functions [Bubeck, 2015]. When we are minimizing a function over a simplex like we are doing (the ℓ_1 ball is replaced by a simplex when negation closure holds (4.17)), sometimes this is the best rate we can achieve for a solution that is k-sparse! This is especially appealing in our case where we are trying to choose a finite number of predictors from an infinite set.

The version of FW presented above is called the *fully corrective* version, this is because we are optimizing over the active set at each iteration as given by (5.14). In contrast, the *vanilla* version of FW has a much simpler update in place of (5.14). It updates the θ by a simple rule as

$$\theta^{(k+1)} = \frac{k}{k+2}\theta^{(k)} + \frac{2}{k+2}s^{(k)}, \tag{5.15}$$

thus eliminating to need to minimize f over $\mathcal{A}^{(k)} \subset \mathcal{D}$, which could potentially be as hard as the original problem.

5.3.1 Frank-Wolfe for Infinite Dimensional Lasso

Now let us see how FW can be applied to infinite dimensional problems (i.e. for measures) like (5.1). Instead of a coefficient vector, we want to find a measure $\boldsymbol{\mu}$ that solves

$$\min_{\boldsymbol{\mu}} \ \mathbf{L}\left(\Phi\boldsymbol{\mu};\mathbf{y}\right) \quad \text{s.t.} \quad \left|\boldsymbol{\mu}\right|(\Omega) \leqslant C. \tag{5.16}$$

The function we are minimizing is $f(\boldsymbol{\mu}) = \mathbf{L}\left(\Phi\boldsymbol{\mu};\mathbf{y}\right)$ and $f : \mathbb{R}^\infty \mapsto \mathbb{R}$ as opposed to $\mathbf{L} : \mathbb{R}^n \mapsto \mathbb{R}$.

At iteration k of FW, given the current estimate $\boldsymbol{\mu}^{(k)}$, this is what the infinite dimensions mean for each of the three FW steps mentioned above:

1. The inner product in the linearization modifies as

$$\begin{aligned}\left\langle \nabla f(\boldsymbol{\mu}^{(k)}), \boldsymbol{\mu} \right\rangle_\infty &= \left\langle \Phi^T \nabla \mathbf{L}\left(\Phi\boldsymbol{\mu}^{(k)};\mathbf{y}\right), \boldsymbol{\mu} \right\rangle_\infty \\ &= \left\langle \nabla \mathbf{L}\left(\Phi\boldsymbol{\mu}^{(k)};\mathbf{y}\right), \Phi\boldsymbol{\mu} \right\rangle_n\end{aligned}$$

The first equality is by the linearity of the operator Φ, defined in (4.5). The second simply moves the linear functional around. (More rigorous proofs can be found in [Bach, 2014, Boyd et al., 2015].) Notably, the ∞-dim inner product reduces to an n-dim one. To ease notation, we define the gradient of the loss with respect to $\Phi\boldsymbol{\mu}$ as

$$\mathbf{g}(\boldsymbol{\mu}) = \nabla_{\mathbf{z}} \mathbf{L}\left(\mathbf{z};\mathbf{y}\right)|_{z=\Phi\boldsymbol{\mu}}. \tag{5.17}$$

giving us

$$\left\langle \nabla f(\boldsymbol{\mu}^{(k)}), \boldsymbol{\mu} \right\rangle_\infty = \left\langle \mathbf{g}(\boldsymbol{\mu}^{(k)}), \Phi\boldsymbol{\mu} \right\rangle_n. \tag{5.18}$$

2. Finding measure $s^{(k)}$ to add to active set:

$$s^{(k)} = \operatorname*{argmin}_{|\boldsymbol{\mu}|(\Omega)=C} \left\langle \mathbf{g}(\boldsymbol{\mu}^{(k)}),\ \Phi\boldsymbol{\mu} \right\rangle \tag{5.19}$$

Note that this is just the minimization of a linear function (an inner product with a constant $\mathbf{g}$) over an ℓ_1 constrained set $\left|\boldsymbol{\mu}\right|(\Omega) \leqslant C$. We have the standard result that the minimum is attained at a vertex of the ℓ_1 ball, meaning $s^{(k)}$ is a point mass measure at $\omega^{(k)}$ with weight $\pm C$. i.e. it is a vertex $\pm C\delta_\omega$ for some $\omega \in \Omega$.

$$s^{(k)} = \operatorname*{argmin}_{\pm C\delta_\omega | \omega\in\Omega} \left\langle \mathbf{g}(\boldsymbol{\mu}^{(k)}), \pm\Phi C\delta_\omega \right\rangle$$

In the next step we are performing a Lasso, so for now we just need to find the $\omega^{(k)}$ that defines $s^{(k)}$

$$\begin{aligned}\omega^{(k)} &= \operatorname*{argmin}_{\omega\in\Omega} \left\langle \mathbf{g}(\boldsymbol{\mu}^{(k)}),\ \pm\Phi\delta_\omega \right\rangle \\ &= \operatorname*{argmin}_{\omega\in\Omega} \left\langle \mathbf{g}(\boldsymbol{\mu}^{(k)}),\ \pm\phi(\mathbf{X};\omega) \right\rangle \\ &= \operatorname*{argmax}_{\omega\in\Omega} \left| \left\langle \mathbf{g}(\boldsymbol{\mu}^{(k)}),\ \phi(\mathbf{X};\omega) \right\rangle \right|\end{aligned} \tag{5.20}$$

3. Now solve Lasso on the active set $\mathcal{A}^{(k)} = \{\omega^{(i)}\}_{i=1}^{k}$ to get

$$\boldsymbol{\mu}^{(k+1)} = \sum_{j=1}^{k} \beta_j \delta_{\omega^{(j)}} \tag{5.21}$$

where,

$$\boldsymbol{\beta} = \operatorname*{argmin}_{\|\boldsymbol{\beta}\|_1 \leqslant C} \mathbf{L}\left(\sum_{j=1}^{k} \beta_j \phi(\mathbf{X};\omega_j); \mathbf{y}\right) \tag{5.22}$$

Note that this is equivalent to finding the best θ on the simplex in (5.14).

In FW, we are effectively building a neural network like the one in Figure 3.3 by adding a node at each iteration. That is, at each iteration, one more atom pops up into the measure. If the subproblem (5.20) can be solved well, we could still enjoy the convergence properties of FW mentioned above, since most losses we consider like MSE, Logit, Poisson, etc. are smooth in $\boldsymbol{\mu}$. Algorithm 2 summarizes this procedure.

Algorithm 2 Frank-Wolfe (with optional Alternating Descent)

Input: $\mathbf{X}, \mathbf{y}$
Choices: $\phi, \Omega, \mathbf{L}, C$
$\boldsymbol{\mu} \leftarrow \mathbf{0}$ ($\boldsymbol{\beta} \leftarrow [\,], \mathcal{A} \leftarrow \{\}$)
repeat
    $\mathbf{g}(\boldsymbol{\mu}) \leftarrow \nabla_{\mathbf{z}} \mathbf{L}(\mathbf{z}; \mathbf{y})\,|_{\mathbf{z}=\Phi\boldsymbol{\mu}}$ ▷ **gradient**
    $\omega^* \leftarrow \operatorname{argmax}_{\omega\in\Omega} |\langle \mathbf{g}(\boldsymbol{\mu}), \phi(\mathbf{X};\omega)\rangle|$ ▷ **best-ω**
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{\omega^*\}$
    $\boldsymbol{\beta} \leftarrow \mathsf{Lasso}(\mathbf{L}(\phi(\mathbf{X};\mathcal{A}); \mathbf{y}), \mathsf{bound}=C)$ ▷ **LassoOn$\mathcal{A}$**
    if AD enabled **then**
        repeat
            for $\omega \in \mathcal{A}$ **do**
                $\omega \leftarrow \omega - \alpha\nabla_\omega \mathbf{L}$ ▷ **backprop-ω**
            $\boldsymbol{\beta} \leftarrow \mathsf{Lasso}(\mathbf{L}(\phi(\mathbf{X};\mathcal{A}); \mathbf{y}), \mathsf{bound}=C)$
        until convergence
until convergence

Subroutines

More details are presented in Algorithm 2 for fitting a convex neural network via the Frank-Wolfe method. Here we use the following subroutines:

1. **`gradient`**

$$\mathbf{g}(\boldsymbol{\mu}) \leftarrow \nabla_{\mathbf{z}} \mathbf{L}\left(\mathbf{z}; \mathbf{y}\right)|_{\mathbf{z}=\Phi\boldsymbol{\mu}}$$

This is the same one from Algorithm 1 for Neural Networks.

2. **`best-`**ω

$$\omega^* \leftarrow \underset{\omega\in\Omega}{\operatorname{argmax}} |\langle \mathbf{g}(\boldsymbol{\mu}), \phi(\mathbf{X}; \omega)\rangle|$$

This subroutine is specialized for (ϕ, Ω), depends on the constant $\mathbf{X}$ and takes $\mathbf{g}$ as input parameter.

3. **`LassoOn`**$\mathcal{A}$ (in bound form)

$$\boldsymbol{\beta} \leftarrow \mathsf{Lasso}(\mathbf{L}\left(\phi(\mathbf{X}; \mathcal{A}); \mathbf{y}\right), \mathsf{bound}{=}C)$$

This is specialized for the loss, depends on the constant $\mathbf{y}$ and takes as input the matrix $\phi(\mathbf{X}; \mathcal{A}) \in \mathbb{R}^{n\times|\mathcal{A}|}$ and C.

4. **`backprop-`**ω This is the same as in Algorithm 1.

5.3.2 Alternating Descent Frank-Wolfe

After k iterations of FW, we have a measure with k atoms in it,

$$\boldsymbol{\mu}^{(k)} = \sum_{i=1}^{k} \mu_j \delta_{\omega^{(j)}}. \tag{5.23}$$

And the loss becomes

$$\mathbf{L}\left(\sum_{j=1}^{k} \mu_j \phi(\mathbf{X}; \omega^{(j)}); \mathbf{y}\right). \tag{5.24}$$

This loss is differentiable with respect to $\{(\mu_j, \omega^{(j)})\}_{j=1}^{n}$. The derivatives are given by given by (5.7). We are not utilizing the fact that the loss of the problem is differentiable. Remember that this differentiability is the key to fitting a NN. The FW algorithm does not leverage this fact at all. For potentially better fitting, we can gradient descend in the μ and ω spaces between the FW iterations to get Algorithm 2 due to [Boyd et al., 2015]. Figure 5.3 gives a pictoral demonstration of this step.

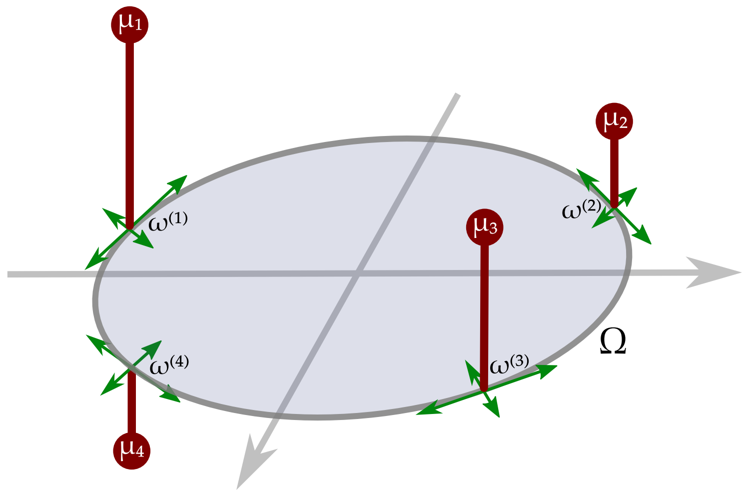

Figure 5.3: Demonstration of Alternating Descent. The current $\boldsymbol{\mu}$ has four atoms located at the edge of say Ω_2. We can move the four atoms around or even change the heights of each atom.

5.4 Stagewise

Now we will examine the Stagewise Algorithm (SW) that can be used to solve ℓ_1 constrained problems approximately [Hastie et al., 2001, Tibshirani, 2015]; and a modification of the same that can be used to solve finite dimensional ℓ_1 constrained problems exactly [Zhao and Yu, 2007]. For the sake of completeness we present the fully general version of stagewise and then specialize it to our problem. In full generality, we want to solve

$$\min_{\theta} f(\theta) \quad \text{s.t.} \quad g(\theta) \leqslant C$$

over a range of values of C, starting from 0 to a point where the constraint is not strict anymore. SW works best for differentiable f and for convex f, g.

Generalized Stagewise takes a step along a direction that most correlates to the gradient of f while still having a small g.

$$\begin{aligned} \theta^{(k)} &= \theta^{(k-1)} + \Delta, \\ \text{where} \quad \Delta &\in \underset{\delta}{\operatorname{argmin}} \left\langle \nabla f(\theta^{(k-1)}), \delta \right\rangle \quad \text{s.t.} \quad g(\delta) \leqslant \epsilon. \end{aligned} \tag{5.25}$$

From this update it is clear that SW (much like FW) minimizes a linearized version of f. But it does so in a small neighbourhood around the current iterate, rather than over the set $g(\theta) \leqslant C$ like FW does. This is the main difference between the two methods.

Usually g is a norm obeying the triangle inequality. Then our problem becomes:

$$\min_{\theta} f(\theta) \quad \text{s.t.} \quad \|\theta\| \leqslant C.$$

In this case, the update has a simple form:

$$\theta^{(k)} = \theta^{(k-1)} - \epsilon\, \partial \left\|\cdot\right\|_* \left(\nabla f(\theta^{(k-1)})\right). \tag{5.26}$$

As the notation suggests, $\partial \left\|\cdot\right\|_*$ is the sub-differential of the dual norm given by

$$\partial \left\|\cdot\right\|_* (a) \in \underset{\|\delta\|_* \leqslant 1}{\operatorname{argmax}} \langle \delta, a \rangle .$$

The next iterate for θ is within an ϵ g-ball around the current:

$$\theta^{(k)} = \theta^{(k-1)} + \epsilon \underset{\|\delta\|_* \leqslant 1}{\operatorname{argmin}} \left\langle \delta, \nabla f(\theta^{(k-1)}) \right\rangle . \tag{5.27}$$

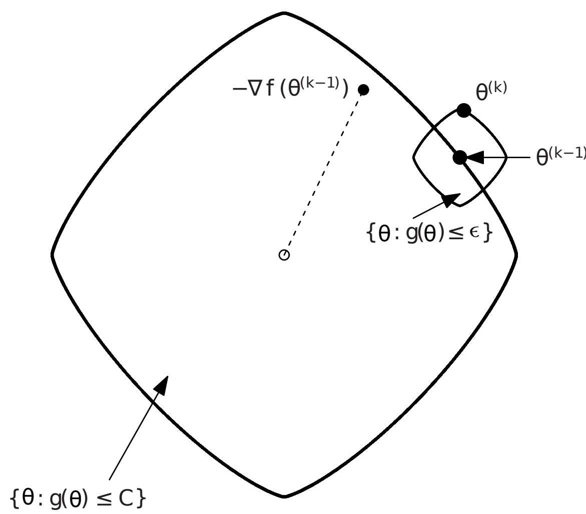

Figure 5.4: Stagewise method in 2D. θ is moved in an ϵ g-ball around its current value, in a direction dictated by the the current gradient. Sourced from [Tibshirani, 2015].

Notice that Stagewise applies a g-based correction to ∇f and steps in that direction. This is akin to Proximal Gradient Descent, which first steps along ∇f and then does a 'prox' correction based on $g \equiv \|\cdot\|$. The PGD update is given by

$$\begin{aligned}
\theta_{\mathsf{PGD}}^{(k)} &= \operatorname{prox}_{\epsilon\|\cdot\|}\left(\theta^{(k-1)} - \nabla f(\theta^{(k-1)})\right) \\
&= \operatorname*{argmin}_{\delta}\left(\epsilon\,\|\delta\| + \frac{1}{2}\left\|\theta^{(k-1)} - \nabla f(\theta^{(k-1)}) - \delta\right\|_2^2\right) \\
&= \operatorname*{argmin}_{\delta}\left(\epsilon\,\|\delta\| + \left\langle \delta, \nabla f(\theta^{(k-1)}) - \theta^{(k-1)} \right\rangle + \frac{1}{2}\,\|\delta\|_2^2\right) \qquad (5.28)
\end{aligned}$$

Thus the prox-update (5.28) is just a shuffled stagewise update (5.27).

5.4.1 Stagewise for Lasso

Let us specialize stagewise for the Lasso problem. Where,

$$\begin{aligned}
f(\boldsymbol{\beta}) &= \mathbf{L}\left(\mathbf{X}\boldsymbol{\beta}; \mathbf{y}\right), \\
g(\boldsymbol{\beta}) &= \|\boldsymbol{\beta}\|_1, \\
\|\nabla f\|_* &= \|\nabla f\|_\infty .
\end{aligned}$$

We want to solve

$$\min_{\boldsymbol{\beta}} \mathbf{L}\left(\mathbf{X}\boldsymbol{\beta}; \mathbf{y}\right) \quad \text{s.t.} \quad \|\boldsymbol{\beta}\|_1 \leqslant C. \qquad (5.29)$$

At each step Stagewise updates $\boldsymbol{\beta}$ (along a cardinal direction e_j) as,

$$\begin{aligned}
\boldsymbol{\beta}^{(k)} &= \boldsymbol{\beta}^{(k-1)} + \epsilon s\, e_{j^*} \\
j^* &\in \operatorname*{argmax}_{j} \left|\nabla_j f(\boldsymbol{\beta}^{(k-1)})\right| \\
&\in \operatorname*{argmax}_{j} \left|\langle \mathbf{X}_j, \mathbf{g}^{(k-1)} \rangle\right| \\
s &= -\operatorname{sign}\langle \mathbf{X}_{j^*}, \mathbf{g}^{(k-1)} \rangle
\end{aligned} \qquad (5.30)$$

Here $\mathbf{g}^{(k)}$ is the current gradient of the loss with respect to the fit given by

$$\mathbf{g}^{(k)} = \nabla_{\mathbf{z}} \mathbf{L}\left(\mathbf{z}; \mathbf{y}\right)|_{\mathbf{z}=\mathbf{X}\boldsymbol{\beta}^{(k)}} .$$

5.4.2 Forward-Backward Stagewise (SW-FB)

We can not use SW to get the exact Lasso path for the above problem. The stagewise paths could differ significantly from the lasso paths when the predictors are highly correlated. To mitigate this, [Zhao and Yu, 2007] propose a modification to the stagewise procedure so that it gives exact lasso paths. The modified version called Forward-Backward Stagewise (SW-FB) involves a backward step.

Before we update $\boldsymbol{\beta}$ as in (5.30), we check to see if the regularized loss

$$\mathbf{L}\left(\mathbf{X}\boldsymbol{\beta};\mathbf{y}\right)+\lambda^{(k)}\left\|\boldsymbol{\beta}\right\|_1$$

can be *reduced* by decreasing $\|\boldsymbol{\beta}\|_1$ by ϵ. If it is indeed possible, we do it. Notice that here we work with the lagrangian formulation of the problem, and λ is calculated at each step as

$$\begin{aligned}\lambda^{(k)}&=\frac{\mathbf{L}\left(\mathbf{X}\boldsymbol{\beta}^{(k)};\mathbf{y}\right)-\mathbf{L}\left(\mathbf{X}\boldsymbol{\beta}^{(k-1)};\mathbf{y}\right)}{\left\|\boldsymbol{\beta}^{(k)}\right\|_1-\left\|\boldsymbol{\beta}^{(k-1)}\right\|_1}\\&=-\frac{\Delta\mathbf{L}^{(k)}}{\epsilon}\end{aligned}$$

As epsilon goes to zero, this method gives paths that are known to converge to the true lasso solution paths (for $p<\infty$).

5.4.3 Stagewise for Infinite Dimensional Lasso

Now we adapt the SW procedure for the infinite dimensional lasso where we are trying to minimize

$$\mathbf{L}\left(\Phi\boldsymbol{\mu};\mathbf{y}\right)\quad\text{s.t.}\quad|\boldsymbol{\mu}|(\Omega)\leqslant C. \tag{5.31}$$

The stage-wise update rule is

$$\begin{aligned}\boldsymbol{\mu}^{(k)}&=\boldsymbol{\mu}^{(k-1)}+\epsilon s\,\delta_{\omega^*}\\\text{where,}\quad\omega^*&\in\underset{\omega\in\Omega}{\operatorname{argmax}}\left|\left\langle\mathbf{g}^{(k-1)},\phi(\mathbf{X};\omega)\right\rangle\right|\quad\text{and}\\s&=-\operatorname{sign}\langle\mathbf{g}^{(k-1)},\phi(\mathbf{X};\omega^*)\rangle.\end{aligned}\tag{5.32}$$

Where again the gradient (or the negative generalized residual) is

$$\mathbf{g}(\boldsymbol{\mu}^{(k)})=\nabla_{\mathbf{z}}\mathbf{L}\left(\mathbf{z};\mathbf{y}\right)|_{\mathbf{z}=\Phi\boldsymbol{\mu}^{(k)}}.$$

Note that the infinite dimensional update (5.32) is equivalent to the finite dimensional one (5.30) modified from a finite $\Omega=[p]$ to an infinite one. Even though in theory the extension is quite straight forward, in practice one should be very careful in the infinite dimensional case. The inner product $\langle\mathbf{g},\phi(\mathbf{X},\omega)\rangle$ as a function of ω is a discrete in the finite dimensional case, but in the infinite dimensional case, it is a continuous one over $\omega\in\Omega$. In the finite dimensional case, the same j^* repeats every few iterations, giving us path algorithms. But in the infinite dimensional case, due to precision issues, we might not pick the same ω^* ever again, and a new ω extremely close to one in the active set $\mathcal{A}$ might get picked at a later iteration, this could keep happening over and over again, resulting in too many predictors entering the model. This means our norm constraint imposition is not working very well to encourage sparsity.

This problem is easily remedied by adding a tolerance. When we are looking for the best ω^* at a given iteration, we first check which of the existing ω's already in the active set is the most favourable. Then we search for the best ω^* as dictated by (5.32). If the new ω^* beats the old ones by at least, say 10^{-4}, we add it to the active set, else we augment the best of the existing ones. This algorithm is presented in pseudo-code in Algorithm 3 with an option for alternating descent presented below.

Alternating Descent Stagewise

As in the case of FW, we observe that SW does not utilize the differentiability of loss w.r.to ω; and just like in FW, we can add that on to SW, using the formula (5.7). Although here the height of the atoms does not change between iterations, it is bound to be multiples of ϵ, only the location of the atoms moves around in Ω, which is less aggressive than AD in the FW case.

Subroutines

The following subroutines are used by the SW Algorithm 3. All these have been already presented for the FW algorithm.

1. **`gradient`**
2. **`best-`**ω
3. **`backprop-`**ω

5.5 Gradient Boosting

While quite often the names Stagewise and Boosting are used interchangeably, we make a distinction here. Stagewise refers to the Generalized Stagewise algorithm above, as specialized to our problem. By boosting we refer to Gradient Boosting [Friedman, 2001] that has other classical forms of boosting like AdaBoost[Freund and Schapire, 1997] as its special cases. Although Gradient Boosting(GB) is always thought of in terms of using regression or classification trees as weak-learners. One can replace trees with any activation ϕ that interests us. Although GB is not originally presented as our model (5.1), it has later been viewed in those terms [Rosset et al., 2004, Friedman, 2012]. Usually in this sense, Ω is thought to have countable number of weak-learners indexed by ω. This makes sense while boosting with trees, as the number of possible trees while being huge is still finite. This is however not the case for ReLUs where Ω is not countable.

Once the boosting problem is formulated as a special case of our problem, we can solve it using SW method. This is what GB is, except with one key difference. While SW increases the complexity

Algorithm 3 Generalized Stagewise

- The flag `FB` specifies if we are using SW-FB. `AD` specifies if we are performing alternating descent in the ω space.
- For simplicity of exposition, we assume Negation Closure (Assumption 1)
- Notate $\mathbf{L}(\boldsymbol{\mu}) = \mathbf{L}(\Phi\boldsymbol{\mu}; \mathbf{y})$ and $\mathbf{g}(\boldsymbol{\mu}) = \nabla_{\mathbf{z}}\mathbf{L}(\mathbf{z}; \mathbf{y})\,|_{\mathbf{z}=\Phi\boldsymbol{\mu}}$

Input: $\mathbf{X}, \mathbf{y}$
Choices: $\phi, \Omega, \mathbf{L}, \epsilon, tol$
$\boldsymbol{\mu} \leftarrow \mathbf{0}$
repeat
▷ Try decrement existing
  $l_-, \omega_- \leftarrow \min, \operatorname{argmin}_{\omega\in\mathcal{A}} \mathbf{L}(\boldsymbol{\mu} - \epsilon\delta_\omega)$
  if $l_- - \texttt{FB} * \lambda\epsilon < \mathbf{L}(\boldsymbol{\mu}) - tol$ **then**
    $\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} - \epsilon\delta_{\omega_-}$
    continue
▷ Try increment existing
  $l_+, \omega_+ \leftarrow \min, \operatorname{argmin}_{\omega\in\mathcal{A}} \mathbf{L}(\boldsymbol{\mu} + \epsilon\delta_\omega)$
▷ Try bringing in new
  $\mathbf{g}(\boldsymbol{\mu}) \leftarrow \nabla_{\mathbf{z}}\mathbf{L}(\mathbf{z}; \mathbf{y})\,|_{\mathbf{z}=\Phi\boldsymbol{\mu}}$ ▷ **`gradient`**
  $\omega^* \leftarrow \operatorname{argmin}_{\omega\in\Omega} \langle \mathbf{g}(\boldsymbol{\mu}), \phi(\mathbf{X}; \omega)\rangle$ ▷ **`best-`**ω
  $l_* \leftarrow \mathbf{L}(\boldsymbol{\mu} + \epsilon\delta_{\omega^*})$
▷ Compare existing vs new
  if $l_+ < l_* + tol$ **then**
    $\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} + \epsilon\delta_{\omega_+}$
  else
    $\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} + \epsilon\delta_{\omega^*}$
▷ Update λ
  $\lambda \leftarrow \min(\lambda, -\Delta\mathbf{L}/\epsilon)$

  if `AD` enabled **then**
    Move $\{\omega_j\}$ around ▷ **`backprop-`**ω
until convergence
return $\boldsymbol{\mu}$

---

of the fit by a maximum of ϵ at each iteration, in GB this restriction is not there. Akin to the SW update (5.30) for the finite case, here it is,

$$\boldsymbol{\beta}^{(k)} = \boldsymbol{\beta}^{(k-1)} + \alpha^{(k)} s\, e_{j^*}, \tag{5.33}$$

where $\alpha^{(k)}$ varies at each step and is chosen by line-search to best minimize the loss greedily. Also, in GB, weak-learners that are in the model once are not changed in any way. As one would expect the greediness would lead to over-fitting and to compensate for it, we damp the step-size $\alpha^{(k)}$ by a multiplicative factor $\gamma < 1$. There could be other differences depending on the implementation but we restrict our attention to these.

The GB update rule for our problem (5.1) is

$$\begin{aligned} \omega^* &\in \operatorname*{argmax}_{\omega\in\Omega} \left|\left\langle \mathbf{g}^{(k-1)}, \phi(\mathbf{X};\omega)\right\rangle\right| \\ \alpha^* &= \operatorname*{argmin}_{\alpha} \mathbf{L}\left(\Phi(\boldsymbol{\mu}^{(k-1)} + \alpha\delta_{\omega^*}); \mathbf{y}\right) \\ &= \operatorname*{argmin}_{\alpha} \mathbf{L}\left(\mathbf{z}^{(k-1)} + \alpha\phi(\mathbf{X};\omega^*); \mathbf{y}\right) \\ \boldsymbol{\mu}^{(k)} &= \boldsymbol{\mu}^{(k-1)} + \gamma\alpha^*\delta_{\omega^*} \end{aligned} \tag{5.34}$$

Notice however in GB applied to trees, the next weak-learner is picked to minimize SSE with the current generalized residual $\mathbf{r}(\boldsymbol{\mu}) = -\mathbf{g}(\boldsymbol{\mu})$,

$$\omega^* \in \operatorname*{argmin}_{\omega\in\Omega} \min_{\rho} \left\|\mathbf{r}(\boldsymbol{\mu}) - \rho\,\phi(\mathbf{X};\omega)\right\|_2^2. \tag{5.35}$$

This is equivalent to our GB update (5.34) when

$$\left\|\phi(\mathbf{X};\omega)\right\|_2 = 1 \;\forall\omega\in\Omega.$$

Which is precisely the definition of our set $\Omega_{\mathbf{X}}$. Which means traditional GB assumes and fixes Ω to be $\Omega_{\mathbf{X}}$ for any activation ϕ. Hence our formulation of GB is much more general. Similar to Stagewise, here also we can do alternating descent between iterations.

Algorithm 4 Gradient Boosting

Input: $\mathbf{X}, \mathbf{y}$
Choices: $\phi, \Omega, \mathbf{L}, \gamma, tol$
$\boldsymbol{\mu} \leftarrow \mathbf{0}$
repeat
  $\mathbf{g}(\boldsymbol{\mu}) \leftarrow \nabla_{\mathbf{z}}\mathbf{L}\left(\mathbf{z};\mathbf{y}\right)|_{\mathbf{z}=\Phi\boldsymbol{\mu}}$
  $\omega^* \leftarrow \operatorname{argmax}_{\omega\in\Omega} |\langle \mathbf{g}(\boldsymbol{\mu}), \phi(\mathbf{X};\omega)\rangle|$ ▷ **best-**ω
  $\alpha^* \leftarrow \operatorname{argmin}_{\alpha} \mathbf{L}(\boldsymbol{\mu} + \alpha\delta_{\omega^*})$ ▷ **LineSearch**
  $\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} + \gamma\alpha^*\delta_{\omega^*}$

  if AD enabled **then**
    Move $\{\omega_j\}$ around ▷ **backprop-**ω
until convergence
return $\boldsymbol{\mu}$

Subroutines

Gradient boosting uses the same subroutines as Stagewise, with one extra.

1. **gradient**
2. **best-**ω
3. **backprop-**ω
4. **line-search**

$$\begin{aligned}\alpha^* &\leftarrow \operatorname*{argmin}_{\alpha} \mathbf{L}\left(\Phi(\boldsymbol{\mu} + \alpha\delta_{\omega^*}); \mathbf{y}\right)\\ &\leftarrow \operatorname*{argmin}_{\alpha} \mathbf{L}\left(\Phi\boldsymbol{\mu} + \alpha\phi(\mathbf{X}; \omega^*); \mathbf{y}\right)\end{aligned}$$

This subroutine is specialized for the loss, it depends on the constant $\mathbf{y}$ and takes as arguments the current fit $\Phi\boldsymbol{\mu}$ and the predictor $\phi(\mathbf{X}; \omega^*)$.

5.6 Comparison

We have seen an assembly of algorithms that try to find good non-linear weak-learners and then combine them to get good fits. In the special case of ReLU like projected activations, we find informative directions of projections of the data. Here we summarize with a comparison the various algorithms.

Projection Pursuit Regression and Hinging Hyperplanes

As background in chapter 3, we saw the historical methods PPR and HH. They solve the problem in a forward step-wise manner. These methods are not trying to solve the problem as we formulated in (5.1), but can still be seen as methods that are trying to find good ensembles that generalize well. These methods come with an option to correct old projections after a new one is added. The only regularization here is early stopping. Both of them employ Gauss-Newton method to find the next predictor, and exploit the differentiability of the loss function with respect to the directions, $\nabla_{\omega_j} L$ in (5.7), to get a better fit on the training data. HH is a sub-class of PPR that uses only Hinges.

Neural Networks

A NN starts with a predetermined number of randomly initialized directions and coefficients performs gradient descent in their space. There is no incremental addition of predictors. They depend solely on $\nabla_{\omega_j} L$ and $\nabla_{\boldsymbol{\beta}} L$ to obtain a fit. The ℓ_1 constraint is imposed via soft-thresholding (9.33) or projection on to the ℓ_1 ball. The NN solves the problem (5.1) for a fixed value of C. A path of solutions can be obtained by using warm-starts.

Frank-Wolfe

FW is a point algorithm for the main problem (5.1), it adds predictors in an iterative fashion for a fixed bound on $\|\boldsymbol{\mu}\|_1$. At each iteration the ℓ_1 constraint is imposed by solving the finite dimensional lasso problem over and again with the ever increasing active set. This is guaranteed to solve our problem, if the subproblem of finding **`best-`**ω can be solved well [Bach, 2014]. FW can optionally use an AD step, where both $\{(\omega_j, \beta_j)\}$ are updated in a cyclical fashion.

Stagewise

SW is a much simpler algorithm that can give an entire path of solutions for increasing values of the regularization parameter. Stagewise is also amenable to incorporating alternating descent. We also have an option to incorporate the exact ℓ_1 modification to get possibly sparser solutions via what is called the forward-backward stagewise.

Gradient Boosting

GB is the greedy cousin of stagewise where the step-size parameter γ is a multiplicative correction to the optimal step-size α found via line-search; as opposed to the SW where ϵ is the fixed step-size. GB also gives a path of solutions.

Table 5.1 summarizes the above conceptual differences between the various methods.

Table 5.1: Comparison between the various first order algorithms. The columns are: Name; Stepwise, Stagewise, Pointwise or Exact algorithm; Choice of norm; Continuous(C), Discrete(D), Jumpy(J) or Not a(No) path; Alternating Descent Yes, No or Optional; Activation; Addition of new predictors. (LARs is presented in Chapter 9.)

Algo.	-wise	Norm	Path	AD	Activation	Addition
PPR	step-	—	D	Y	Any	Gauss-Newton
HH	step-	—	D	Y	Hinge	Gauss-Newton
NN	point	$\ell_{0/1}$	No	Y	Projected	Random
FW	point	ℓ_1	No	O	Any	Chapter 6
SW	stage-	ℓ_1	C	O	Any	Chapter 6
GB	stage-	ℓ_1	J	O	Any	Chapter 6
LARS	exact	ℓ_1	C	N	Any	Section 9.1.3

5.6.1 Hyper Parameters

Tuning hyper-parameters is a big part of Machine Learning. Some algorithms like Neural Networks are notorious for the number of hyper-parameters that need to be tuned. All the methods we are going to consider need a grid of values that the series of fits are being requested. For FW we simply run the point algorithm on a pre-specified grid. For SW, the grid is specified indirectly via the number of steps to take and the size of each step, ϵ. In GB, too the grid is specified indirectly, via the number of steps. Further at each step, GB greedily chooses to add as much of the new predictor as it can, this needs to be damped by specifying a damping parameter also called the learning rate. Neural Networks not only need the specification of the grid, in addition we need to specify

- Step size for the ω space
- Step size for the β space (coefficients of predictors)
- The noise distribution of the random initial values of $\{(\omega_j, \beta_j)\}$
- Number of nodes in the hidden layer
- Initial step-size and the annealing routine
- Number of iterations

These hyper-parameters are tabulated in Table 5.2. In addition to these for a given problem, we need to specify the loss $\mathbf{L}$, the activation of choice ϕ and the constraint set Ω. These might further have their own hyper-parameters. Like, in the case of Huber loss, we need to specify the switching threshold, similarly for the Huber activation function. For Ω, we need to specify the norm, which is usually unity. But say for tree activations, we need to specify the maximum depth of the tree and the pruning parameters.

Table 5.2: Hyper-parameters for the various first order algorithms.

Method	Hyper-Parameters
Frank-Wolfe	Grid
Stagewise	Grid (Number of steps $\times \epsilon$)
Boosting	Number of steps, Damping
Neural Network	Grid, Step-size for ω, Step-size for β, Number of nodes, Random initialization parameters, Step-size annealing, Iterations, etc.

5.6.2 Approximation to ℓ_1 Minimization

Our pedagogical aim is to study the relations between various algorithms like Frank-Wolfe (used primarily in the engineering field), Gradient Boosting (developed in Machine Learning) and Stagewise methods (from Statistics) as it applies to our Sparse Inverse Problem (5.1). On the practical side we want to develop a good ensemble method using the underlying principles of good machine learning techniques like Neural Networks and Boosting, which motivates us to propose (5.1), simultaneously a Neural Network and a Boosting framework. If we are interested in a classification problem, we use a good surrogate loss $\mathbf{L}$ for the discontinuous misclassification loss, which is our true loss of interest.

In summary, we are not interested in minimizing our ℓ_1 problem but on one side we are interested in the connections it enables us to see, and on the other hand get sparse ensembles that generalize well. But it still is of value to see how each of the three main methods we proposed try to solve the main lasso problem (5.1).

Frank-Wolfe tries to minimize the convex problem on the domain defined by the bound on the ℓ_1 norm of the coefficient measure. As the number of atoms increases the optimization error decreases as $O(1/k)$. This is very good and for some problem the worst-case rate [Jaggi, 2013, Bach, 2014]. But this hinges on the fact that the hard subproblem (5.20) can be solved well. Given that the subproblem for the activation

$$\phi_\alpha(\mathbf{x}; \mathbf{u}, b) = (\mathbf{x} \cdot \mathbf{u} + b)^\alpha$$

is known to be NP-hard for $\alpha = 0$, and it is conjectured to be so for $\alpha = 1$. We can not give theoretical guarantees of optimality. But given that we have very good algorithms to solve the hard sub-problems for ReLUs with Ω_2 or $\Omega_{\mathbf{X}}$, we still can say that FW is directly solving the ℓ_1 problem.

It is well know that in finite dimensions, **Stagewise** solutions approach that of the Lasso under the positive cone constraint [Efron et al., 2004]. Further, even without that condition, for finite p, [Zhao and Yu, 2007] prove that the Forward-Backward version converges to the Lasso solution. But in our case, p is infinity and we know of no proofs. But again, we can be confident that the approximation is reasonable with SW-FB.

Although **Gradient Boosting** is motivated as an approximate algorithm to solve the ℓ_1 problem (5.1) in [Friedman, 2012], the arbitrariness of the learning rate selection (which is indispensable) makes it harder to make an claims, and we do not know of any results that try to address this question.

The way the **Neural Networks** formulate the ensemble problem is non-convex as the parameters of the weak-learners are treated as unknowns (as opposed to the convex formulation where all the possible predictors are put into the model). For a very long time, NNs have been thought to be finding good local minima, but recent works have shown that these local minima found via stochastic gradient descent (SGD) indeed have desirable properties. [Mei et al., 2018, Wei et al., 2018] show that SGD goes to the max-margin classifier just like Stagewise does [Rosset et al., 2004].

Chapter 6

Next Best Predictor: The Hard Subproblem

We have seen that Frank-Wolfe, Stagewise and Gradient boosting all involve the step where we want to solve:

$$\omega^* = \operatorname*{argmin}_{\omega \in \Omega} \langle \mathbf{g}, \phi(\mathbf{X}; \omega) \rangle . \tag{6.1}$$

We call this problem the subroutine **best-**ω. This is the step where we try to minimize the linearization of the function $\mathbf{L}\left(\Phi \boldsymbol{\mu}^{(k)}; \mathbf{y}\right)$ at the current iterate $\boldsymbol{\mu}^{(k)}$. We arrive at this step variously for the various algorithms, but this step stays the same. Only the $\boldsymbol{\mu}^{(k)}$ changes. Here $\mathbf{g}$ represents the gradient of the loss with respect to the fit $\mathbf{z}$

$$\mathbf{g}(\boldsymbol{\mu}^{(k)}) = \nabla_{\mathbf{z}} \mathbf{L}\left(\mathbf{z}; \mathbf{y}\right) |_{\mathbf{z} = \Phi \boldsymbol{\mu}^{(k)}} . \tag{6.2}$$

Without assuming negation closure, we would want to find

$$\begin{aligned} \omega^* &= \operatorname*{argmin}_{\omega \in \Omega} \langle \pm \phi(\mathbf{X}; \omega), \mathbf{g} \rangle , \\ \text{or} \quad \omega^* &= \operatorname*{argmax}_{\omega \in \Omega} \left| \langle \mathbf{g}, \phi(\mathbf{X}; \omega) \rangle \right| . \end{aligned} \tag{6.3}$$

That is we solve (6.1) once for $+\mathbf{g}$ and once for $-\mathbf{g}$. Henceforth, we will discuss how to solve (6.1) for a given vector $\pm\mathbf{g}$. This could be a hard problem and the success of such a search depends mostly on the nature of ϕ and to some extent on Ω. We will discuss this problem for various choices of (ϕ, Ω) in the following sections of this chapter.

6.1 Any ϕ, $\Omega_{\mathbf{X}}$

As background we present the boosting example. Where the activation function can be anything but is usually a classification or regression tree (parameterized by ω — the splits, and terminal values). As mentioned previously, in boosting there is an implicit assumption that all weak-learner trees under consideration 'weigh' the same:

$$\forall \omega \in \Omega, \quad \|\phi(\mathbf{X};\omega)\|_2 = 1.$$

This is our definition of $\Omega_{\mathbf{X}}$. Representing $\mathbf{z} = \phi(\mathbf{X};\omega)$, noting $\|\mathbf{z}\|_2 = 1$, consider approximating $\mathbf{g}$ by a multiple of $\mathbf{z}$.

$$\begin{aligned}\min_{\rho} \|\mathbf{g} - \rho\mathbf{z}\|_2^2 &= \min_{\rho} \|\mathbf{g}\|_2^2 + \rho^2 - 2\rho \langle \mathbf{z}, \mathbf{g} \rangle \\ &= \|\mathbf{g}\|_2^2 - \langle \mathbf{z}, \mathbf{g} \rangle^2\end{aligned}$$

This is obtained when $\rho^* = \langle \mathbf{z}, \mathbf{g} \rangle$. Meaning

$$\operatorname*{argmin}_{\|\mathbf{z}\| \leqslant 1} \min_{\rho} \|\mathbf{g} - \rho\mathbf{z}\|_2^2 = \operatorname*{argmax}_{\|\mathbf{z}\| \leqslant 1} |\langle \mathbf{z}, \mathbf{g} \rangle| \,.$$

Thus our problem (6.3) becomes the **nonlinear least squares** problem:

$$\hat{\omega}^* = \operatorname*{argmin}_{\omega} \|\mathbf{g} - \phi(\mathbf{X};\omega)\|_2^2 \tag{6.4}$$

Then to get our optimal $\omega^* \in \Omega_{\mathbf{X}}$ we just scale the predictor down as

$$\phi(\mathbf{X};\omega^*) = \frac{\phi(\mathbf{X};\hat{\omega}^*)}{\|\phi(\mathbf{X};\hat{\omega}^*)\|_2}.$$

In general for trees, the problem (6.4) is NP hard. But the nonlinear least squares problem is very well understood. And in case of trees, it is obvious from their popularity that the hardness has not hindered its employment.

6.2 ReLU, $\Omega_{\mathbf{X}}$

We now specialize the above for $\phi \equiv$ ReLU. Using ReLU as our activation and limiting the ℓ_2-norm of the predictors, we want to solve the hard subproblem (6.3),

$$\max_{(\mathbf{u},b)} |\langle \mathbf{g}, (\mathbf{X}\mathbf{u}{+}b)_+ \rangle| \quad \text{s.t.} \quad \|(\mathbf{X}\mathbf{u}{+}b)_+\|_2 \leqslant 1. \tag{6.5}$$

It is obvious that for the maximizing predictor, $\|(\mathbf{X}\mathbf{u}+b)_+\|_2 = 1$; this is because the ReLU is positively homogeneous and hence the inner-product scales linearly in $(\mathbf{u}, b)$. i.e. for any $\rho \geq 0$

$$\big\langle(\mathbf{X}\rho\mathbf{u} + \rho b)_+, \mathbf{g}\big\rangle = \rho\ \big\langle(\mathbf{X}\mathbf{u}+b)_+, \mathbf{g}\big\rangle. \tag{6.6}$$

Hence we can fix the norm of the predictor to unity. Similar to the above case with general ϕ, our problem boils down to the NLLS problem as (6.4)

$$(\hat{\mathbf{u}}, \hat{b}) = \operatorname*{argmin}_{(\mathbf{u},b)} \|\pm\mathbf{g} - (\mathbf{X}\mathbf{u}+b)_+\|_2^2 . \tag{6.7}$$

The $\pm$ is because the ReLU (as defined here) is not negativity closed as in Assumption 1, and it is only *positively* homogeneous, unlike trees which satisfy Assumption 1. Once we solve the above, by the homogeneity of the ReLU, the optimum $(\mathbf{u}, b)$ is obtained by scaling as

$$(\mathbf{u}^*, b^*) = \frac{(\hat{\mathbf{u}},\ \hat{b})}{\Big\|(\mathbf{X}\hat{\mathbf{u}}^* + \hat{b}^*)_+\Big\|_2}.$$

Gauss-Newton

Solving (6.7) (for $+\mathbf{g}$ wlog) is a well-studied albeit non-convex problem. We will use the traditional Gauss-Newton method to solve this NLLS problem iteratively [Gill et al., 1981]. Linearizing $(\mathbf{X}\mathbf{u}+b)_+$ around the current estimate, say $(\mathbf{u}_0, b_0)$, the objective becomes

$$\min \Big\|\big(\mathbf{g}-(\mathbf{X}\mathbf{u}_0+b_0)_+\big) - \big(\mathbf{J}_0(\mathbf{u}-\mathbf{u}_0)+(b-b_0)\big)\Big\|_2^2 , \tag{6.8}$$

where $\mathbf{J}_0$ is the the Jacobian of $(\mathbf{X}\mathbf{u}+b)_+$ w.r.t $\mathbf{u}$ at $(\mathbf{u}_0, b_0)$, it is given by (5.8)

$$\mathbf{J}_0 = \operatorname{diag}\big[\mathbb{I}\left(\mathbf{X}\mathbf{u}_0+b_0 \geqslant 0\right)\big]\mathbf{X}. \tag{6.9}$$

Simply put, $\mathbf{J}_0$ has those rows of $\mathbf{X}$ set to zero where the ReLU is inactive. The linearized objective can be rewritten in terms of $\delta_{\mathbf{u}} = \mathbf{u} - \mathbf{u}_0,\ \delta_b = b - b_0$ as

$$\min \big\|\mathbf{g}_0 - \big(\mathbf{J}_0\delta_{\mathbf{u}}+\delta_b\big)\big\|_2^2 , \tag{6.10}$$

where $\mathbf{g}_0 = \mathbf{g}-(\mathbf{X}\mathbf{u}_0+b_0)_+$. This is a straight forward least squares problem in $(\delta_{\mathbf{u}}, \delta_b)$. We repeatedly minimize this partial linearization to get our solution. In practice, in the interest of stability, we also apply a trust region technique ala Levenberg-Marquardt and add the regularization $\lambda\,\|\delta_{\mathbf{u}}\|$ to this local objective. This gives us the ridge regression problem.

$$\min \big\|\mathbf{g}_0 - \big(\mathbf{J}_0\delta_{\mathbf{u}}+\delta_b\big)\big\|_2^2 + \lambda\,\|\delta_{\mathbf{u}}\|_2^2 . \tag{6.11}$$

At each iteration, we keep updating the λ depending on how good our linerization is compared to the original objective. Algorithm 5 summarizes this process to solve (6.5).

Algorithm 5 Gauss-Newton for ReLU and $\Omega_{\mathbf{X}}$

Input: $\mathbf{X}, \mathbf{g}$
Initialize: $(\mathbf{u}, b) = (\mathbf{0}, 0)$
for all $\mathbf{g} \in \{\mathbf{g}(\boldsymbol{\mu}), -\mathbf{g}(\boldsymbol{\mu})\}$ **do**
  repeat
    $\mathbf{g} \leftarrow \mathbf{g} - (\mathbf{X}\mathbf{u}+b)_+$
    $\mathbf{J} \leftarrow \operatorname{diag}[\mathbb{I}((\mathbf{X}\mathbf{u}+b)_+ > 0)]\mathbf{X}$
    $(\mathbf{u}, b) \leftarrow (\mathbf{u}, b) + \operatorname{argmin} \left\|\mathbf{g} - \left(\mathbf{J}\delta_{\mathbf{u}}+\delta_b\right)\right\|_2^2 + \lambda \left\|\delta_{\mathbf{u}}\right\|_2^2$ ▷ Ridge Reg.
    Increase/decrease trust λ based on current objective
  until convergence
$(\mathbf{u}, b) \leftarrow (\mathbf{u}, b) / \left\|(\mathbf{X}\mathbf{u}+b)_+\right\|_2$
Return: Better $(\mathbf{u}, b)$ for $\pm\mathbf{g}(\boldsymbol{\mu})$

6.3 ReLU, Ω_2

Now we use ReLUs whose directions of projection are unit vectors. Here the subproblem of finding the best ω is

$$(\mathbf{u}^*, b^*) = \underset{(\mathbf{u},b)}{\operatorname{argmin}} \langle \pm\mathbf{g}, (\mathbf{X}\mathbf{u}+b)_+ \rangle \quad \text{s.t.} \quad \|\mathbf{u}\|_2 \leqslant 1, |b| \leqslant 1 \tag{6.12}$$

We solve this problem parallely for $+\mathbf{g}$ and $-\mathbf{g}$. We solve it via iterative linearization and minimization. Given that Frank-Wolfe is an iterative minimization technique, we use it here (to solve the subproblem!). At the current iterate $(\mathbf{u}^{(k)}, b^{(k)})$, the ReLU is just a linear function where it is active and zero where inactive. So the linearized problem is

$$(\mathbf{u}^*, b^*) = \underset{(\mathbf{u},b)}{\operatorname{argmin}} \left\langle \mathbf{g}, (\mathbf{J}^{(k)}\mathbf{u}^{(k)} + \mathbb{1}^{(k)} b^{(k)}) \right\rangle \quad \text{s.t.} \quad \|\mathbf{u}\|_2 \leqslant 1, |b| \leqslant 1, \tag{6.13}$$

where $\mathbb{1} = \mathbb{I}(\mathbf{X}\mathbf{u}+b \geqslant 0)$ is the active sample indicator and similarly $\mathbf{J}_0 = \operatorname{diag}(\mathbb{1})\,\mathbf{X}$ is the jacobian with the inactive rows set to zero. The problem separates as

$$\begin{aligned}
\mathbf{u}^* &= \underset{\mathbf{u}}{\operatorname{argmin}} \left\langle \mathbf{J}^{(k)T}\mathbf{g}, \mathbf{u}^{(k)} \right\rangle \quad \text{s.t.} \quad \|\mathbf{u}\|_2 \leqslant 1 \\
&= -\frac{\mathbf{J}^{(k)T}\mathbf{g}}{\left\|\mathbf{J}^{(k)T}\mathbf{g}\right\|_2} \\
b^* &= \underset{b}{\operatorname{argmin}}(b\,\mathbb{1}^{(k)} \cdot \mathbf{g}) \quad \text{s.t.} \quad |b| \leqslant 1 \\
&= -\operatorname{sign}(\mathbb{1}^{(k)} \cdot \mathbf{g}).
\end{aligned}$$

We add the above $(\mathbf{u}^*, b^*)$ to the *vanilla* Frank-Wolfe update (as defined in (5.15)) to get the next iterate:

$$\mathbf{u}^{(k+1)} = \frac{k}{k+2}\mathbf{u}^{(k)} + \frac{2}{k+2}\mathbf{u}^*$$
$$b^{(k+1)} = \frac{k}{k+2}b^{(k)} + \frac{2}{k+2}b^*$$

Then we project $(\mathbf{u}, b)$ on to the boundary

$$\mathbf{u}^{(k+1)} \leftarrow \frac{\mathbf{u}^{(k+1)}}{\left\|\mathbf{u}^{(k+1)}\right\|_2}$$
$$b^{(k+1)} \leftarrow \operatorname{sign}(b^{(k+1)})$$

In practice however, the constraint $|b| \leqslant 1$ is rather arbitrary. It should indeed depend on the distribution of $\mathbf{x}$, which is unknown. And picking $b = \pm 1$ (or any other constant) seems to disrupt the choice of the corresponding $\mathbf{u}$ as the bias b changes the set of $\mathbf{x}_i$ for which the ReLU is active. So instead we modify the algorithm slightly, so that the choice of b depends on $\mathbf{u}$. Concatenating $\omega = [b\ \mathbf{u}]$, $\tilde{\mathbf{X}} = [\mathbf{1}\ \mathbf{X}]$ and $\tilde{\mathbf{J}} = [\mathbb{1}\ \mathbf{J}]$.

$$\omega^* = [b^*\ \mathbf{u}^*] = -\frac{\tilde{\mathbf{J}}^{(k)T}\mathbf{g}}{\left\|\mathbf{J}^{(k)T}\mathbf{g}\right\|_2} \tag{6.14}$$
$$\approx \operatorname*{argmin}_{\omega} \left\langle \tilde{\mathbf{J}}^{(k)T}\mathbf{g}, \omega^{(k)} \right\rangle \quad \text{s.t.} \quad \|\mathbf{u}\|_2 \leqslant 1 \tag{6.15}$$

This update means that $\tilde{\mathbf{J}}^{(k)T}\mathbf{g}$ gives us the $[b^*\ \mathbf{u}^*]$ with the best set of $\{\mathbf{x}_i\}$ that are active. Then we scale them without changing that active set to normalize just $\mathbf{u}^*$ and allow b to take the appropriate value to keep the active set the same.

Then we update using the Frank-Wolfe step as above, and scale so that only $\mathbf{u}$ is normalized to have unit norm.

$$(\mathbf{u}^{(k+1)}, b^{(k+1)}) \leftarrow \frac{(\mathbf{u}^{(k+1)}, b^{(k+1)})}{\left\|\mathbf{u}^{(k+1)}\right\|_2} \tag{6.16}$$

Algorithm 6 summarizes this process to solve (6.12).

6.4 ReLU, Ω_1

Although a nature restriction on the ReLU is to ask for the directions of projection to be unit vectors, in the interest of obtaining spare and interpretable directions, we might choose to restrict the ℓ_1 norm of the directions rather than the ℓ_2 norm. Here the subproblem of finding the best ω is

$$(\mathbf{u}^*, b^*) = \operatorname*{argmin}_{(\mathbf{u}, b)} \langle \pm\mathbf{g}, (\mathbf{X}\mathbf{u}+b)_+ \rangle \quad \text{s.t.} \quad \|\mathbf{u}\|_1 \leqslant 1, |b| \leqslant 1 \tag{6.17}$$

Algorithm 6 Frank-Wolfe to find `best-`ω for ReLU and Ω_2

Input: $\mathbf{X}, \mathbf{g}(\boldsymbol{\mu})$
Initialize: $(\mathbf{u}, b) \leftarrow (\mathbf{0}, 0)$
for all $\mathbf{g} \in \{\mathbf{g}(\boldsymbol{\mu}), -\mathbf{g}(\boldsymbol{\mu})\}$ **do**
  repeat
    $\mathbb{1} \leftarrow \mathbb{I}(\mathbf{X}\mathbf{u}+b \geqslant 0)$
    $\mathbf{J} \leftarrow \operatorname{diag}(\mathbb{1})\,\mathbf{X}$
    $\tilde{\mathbf{J}} \leftarrow [\mathbb{1}\ \mathbf{J}]$
    $[b^*\ \mathbf{u}^*] \leftarrow -\tilde{\mathbf{J}}^T\mathbf{g}/\left\|\mathbf{J}^T\mathbf{g}\right\|$
    $(\mathbf{u}, b) \leftarrow \frac{k}{k+2}(\mathbf{u}, b) + \frac{2}{k+2}(\mathbf{u}^*, b^*)$
    $(\mathbf{u}, b) \leftarrow (\mathbf{u}, b)/\left\|\mathbf{u}\right\|$
  until convergence
Return: Better $(\mathbf{u}, b)$ for $\pm\mathbf{g}(\boldsymbol{\mu})$

We solve this problem parallelly for $+\mathbf{g}$ and $-\mathbf{g}$. We solve it via iterative linearization and minimization. Once again we resort to finding a stationery point for iterative linearization. Introducing the Jacobian we have the linearization at a given point as,

$$(\mathbf{u}^*, b^*) = \underset{(\mathbf{u},b)}{\operatorname{argmin}} \left\langle \mathbf{g}, (\mathbf{J}^{(k)}\mathbf{u}^{(k)} + \mathbb{1}^{(k)}b^{(k)}) \right\rangle \quad \text{s.t.} \quad \left\|\mathbf{u}\right\|_1 \leqslant 1, |b| \leqslant 1, \tag{6.18}$$

The problem separates as

$$\begin{aligned} \mathbf{u}^* &= \underset{\mathbf{u}}{\operatorname{argmin}} \left\langle \mathbf{J}^{(k)T}\mathbf{g}, \mathbf{u}^{(k)} \right\rangle \quad \text{s.t.} \quad \left\|\mathbf{u}\right\|_1 \leqslant 1 \\ &= -e_{j^*} \\ \text{where} \\ j^* &= \underset{j}{\operatorname{argmax}} \left\langle \mathbf{J}^{(k)}_{\cdot j}, \mathbf{g} \right\rangle \\ \text{and} \\ b^* &= \underset{b}{\operatorname{argmin}} (b\,\mathbb{1}^{(k)} \cdot \mathbf{g}) \quad \text{s.t.} \quad |b| \leqslant 1 \\ &= -\operatorname{sign}(\mathbb{1}^{(k)} \cdot \mathbf{g}). \end{aligned}$$

Hence, where ever the linearization is taken, it is maximized by placing all the mass of $\mathbf{u}$ in the co-ordinate that has the highest $\langle \mathbf{J}^{(k)}_{\cdot j}, \mathbf{g} \rangle$.

So any iterative lineraization algorithm will terminate in a vertex of the ℓ_1 ball. So we choose $\mathbf{u}$ to be along the cardinal directions (and negations), and b to be ± 1, find the corresponding active set of datapoints as in (6.9) and pick the vertex with the best objective. Alternatively we start at $\mathbf{u} = \mathbf{0}$, go to a vertex, update the active set and repeat until we hit a stationary vertex. Note that this choice of Ω gives directions that are active only in one co-ordinate, results in a Generalized

Additive Model; and thus seems to have diminished appeal.

Given the nature of the function $\langle \mathbf{g}, (\mathbf{X}\mathbf{u})_+ \rangle$, we do not think any algorithm works well with a restriction of $\|\mathbf{u}\|_1 \leqslant 1$ (the inclusion of b only makes it worse). This inner product is almost a linear function of $\mathbf{u}$. It scales linearly along rays of $\mathbf{u}$, and hence points that are the farthest from the origin (in an ℓ_2 sense) are most preferred. Hence in the vast majority of cases, we suspect that the extremum of $\langle \mathbf{g}, (\mathbf{X}\mathbf{u})_+ \rangle$ lies at a vertex, leading to our algorithms (FW, SW, GB) ending up being generalized additive models. We still present Ω_1 here, as it might be of interest in conjunction with the second order methods discussed in chapter 9.

Chapter 7

Simulation Studies

We now present a simulation study to see how each of the above first order methods — FW, SW, SW-FB, GB and NN with or without AD — compare to one another. We start by noting that there are a lot of choices that are to be made while designing a simulation study, and each one of them will effect the relative performance of the methods. These choices include, the size of training data n, the true target function, the magnitude and the distribution of the noise, the dimensionality of the predictors p, the distribution of the predictors, and the nature of the problem itself.

On top of the above parameters that apply to a simulation study of any method, in our case, more specifically, we need to chose the pair (ϕ, Ω) too. This will further influence the performance of each of the methods, and more importantly the performance of the methods relative to an established algorithm like XGBoost with Trees.

Above all, the nature of the true target function is seldom known in machine learning applications. We need to chose a class of target functions that are rich enough to represent real-world data. Additionally, we need to run multiple simulations and aggregate the results to get meaningful comparisons.

7.1 Problem Instance Generation

We generate multiple problem instances from a class of problems and consider the median of the test error rate to be a reliable indicator of performance. A problem instance is specified by the data $(\mathbf{X}, \mathbf{y})$. While $\mathbf{X}$ is easily generated, $\mathbf{y}$ as a function of $\mathbf{X}$ (the target function, plus noise) is to be generated carefully. Since the choice of target function affects the performance of our methods relative to existing ones, we go with the model from [Friedman, 2001]. This avoids any bias of the target function being easily represented via ReLUs, since the model predates the use of ReLUs in Neural Networks. We start by presenting how we generate $\mathbf{y}$ as a random function of $\mathbf{X}$.

Random Function

Our true target function is a sum of gaussians, taking the form

$$f(\mathbf{x}) = \sum_{m=1}^{10} a_m h_m(\mathbf{x}_{A_m}). \tag{7.1}$$

The coefficients $\{a_m\}$ are randomly generated from the uniform distribution $U[-1, 1]$. Each h_m is a function of a randomly selected subset $A_m \subset A \subset [p]$ of $\mathbf{x}$. Initially A is picked to contain half of the p variables. These $p/2$ variables contained in A are the "good" predictors that will be used in the target function, the rest do not contribute to the target function. This reflects the real-world situations with a lot of spurious variables, that do not have any meaningful connection to the predicted variable.

Each summand further uses a subset of these good variables. The size of each of the subsets A_m is itself a random variable $\lceil \text{Exp}(2) \rceil$ (i.e., an exponential random variable rounded up to the next integer). Thus each summand is a function of two to four of the good variables, making it an interaction among those variables.

Each h_m is a gaussian function (acting on a subset of $\mathbf{x}$, $\breve{\mathbf{x}} := \mathbf{x}_{A_m}$) given by

$$h_m(\breve{\mathbf{x}}) = \exp\left(-\tfrac{1}{2}[(\breve{\mathbf{x}} - \mu_m)^T V_m (\breve{\mathbf{x}} - \mu_m)]\right), \tag{7.2}$$

where each of the mean vectors μ_m, supported on A_m, are generated from the same distribution as that of $\mathbf{x}_{A_m}$. The covariance matrix V_m is also randomly generated as

$$V_m = U_m D_m U_m^T.$$

U_m is a random orthonormal matrix on the Haar measure. The diagonal elements of D_m are iid as squares of uniform random variables $U[1, 2]$.

Data Distribution

For all our studies we use auto-regressive gaussian X, we pick the mean to be zero, the (co)variance is given by the auto regressive matrix $\Sigma_{ij} = \rho^{|i-j|}$. We fix $\rho = .3$. We have $p = 21$. We have 2000 training samples and 5000 test samples. We generate the target function as described above. Then we add gaussian noise, $y = f(\mathbf{x}) + \epsilon$, to have an SNR of 2. i.e. $\text{var}(f(\mathbf{x}))/\text{var}(\epsilon) = 2$. When we are using the above $\mathbf{y}$ in a regression setting, the bayes error is 0.5 and the null error rate is 1.5 (relative to the strength of the the noise free target).

7.2 Methods

For our main study, we pick the regression problem with $\mathbf{L} \equiv MSE$, then in section 7.4, we will pick $\mathbf{L} \equiv Logit$. Our choice of non-linearity is the ReLU. We choose the ReLU because of its popularity and also because it has been well studied [Bach, 2014]. We choose the most natural constraint set Ω_2 that goes well with the ReLU as seen as a positive part of a *projection*. The data $(\mathbf{X}, \mathbf{y})$ are generated as mentioned above. Then we implement the following methods with the described hyper-parameters to find the best $\boldsymbol{\mu}$ for our main problem (4.6).

Frank-Wolfe: The only parameter for the FW algorithm is the gird of values for the norm of $\boldsymbol{\mu}$. We pick the values of the bound C to be in $\{1, 1.5, 2, 3, 4, 6, 8, 12, 16, 24, 32\}$ to get a series of fits.

Gradient Boosting: Here we have one important hyper-parameter. That is the damping rate (learning rate) γ from the GB update rule (5.34). The fit obtained is sensitive to this choice, both in terms of the generalization error and in terms of the cardinality of $\boldsymbol{\mu}$. We pick $\gamma = 1.$, to give GB a chance to look decent on the cardinality front. Any lower value and GB just has too many nodes/atoms in the final fit to be of interest.

Stagewise and Stagewise-FB: These methods do not have any serious hyper-parameters. The step length needs to be sufficiently small and we pick $\epsilon = \frac{1}{16}$. The generalization error deteriorates for very high ϵ, but the overall computation is quicker.

Neural Networks: We know that the NN needs to have at most n node, which is the number of training samples. But in practice sparsity eliminates most of them. So we start with a much smaller number and hope that would be enough. In our case we took $m = 300$, based on results from the FW algorithm; while this is kind of cheating, we did not want to leave the value of m to be arbitrary, as is neural networks already take long times to fit. We use the same series of C's as we did in FW. There are many more complications to fitting neural networks, like the initial random values for the parameters, the step-sizes for each of the layers, etc. These will be discussed further later on, but now we present the results for a configuration of training parameters that actually work.

7.3 Regression Simulation

Given the wide variety of methods that we have discussed and implemented, we present them in sets of comparable algorithms. We start with the three main ones of interest Frank-Wolfe, Stagewise and Neural Networks.

7.3.1 FW, SW, NN

The results are summarized in Figure 7.1. We can see that as the flexibility of the fit increases with $\|\boldsymbol{\mu}\|_1$, the cardinality increases linearly (at a slightly higher rate for the Neural Network). We will study the cardinality/frugality of the algorithms later, for now we focus on performance as measured by the Test Error Rate. Although test error rates are variable and depend on the true function class, etc., given that we are considering the median of 25 problem instances (target functions), we will take the median test error at face value. First we note that the three methods do not differ very significantly, but the neural network achieves an error rate that is 10% better than Frank-Wolfe and 16% better than the stagewise methods (relative to the Bayes error). Although the NN seems like the clear winner, we will see how alternating descent closes that gap and also see how training a NN is much harder than FW, which in turn is harder than SW.

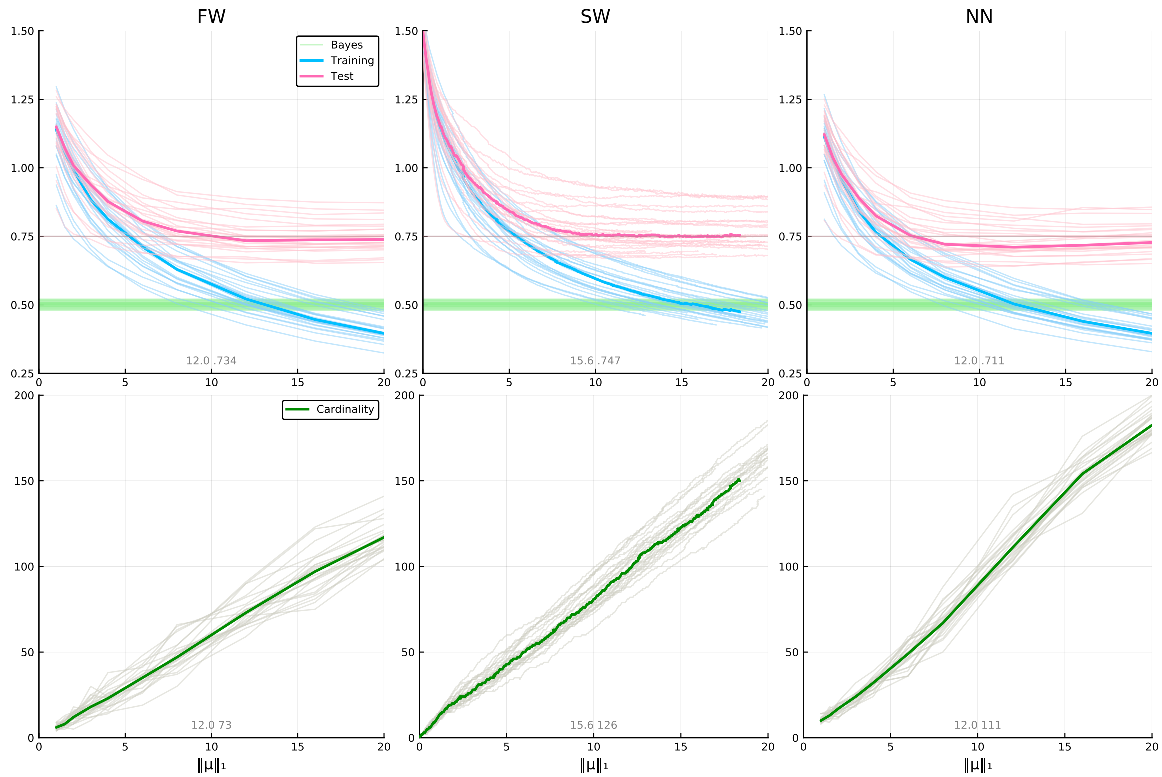

Figure 7.1: Top row: Training, Test and Bayes Errors for FW, SW and NN. Medain of 25 runs is highlighted. Bottom row: Increasing cardinality with $\|\boldsymbol{\mu}\|_1$. FW achieves its best test error of .734 with 73 ReLUs; SW .747 with 126; NN .711 with 111. The NN does better than the other two, and its cardinality rises sharper than that of the others.

7.3.2 GB, SW, SW-FB

We will now consider the three very related methods that perform various forms of gradient boosting. What we refer to GB is gradient boosting in its most popular form (as detailed in Section 5.5). We will also consider the effect of including the modification of [Zhao and Yu, 2007] to the Stagewise algorithms as detailed in Section 5.4.2. This modification achieves the same training error for a smaller value of $\|\boldsymbol{\mu}\|_1$. That is, it solves the infinite dimensional lasso more closely.

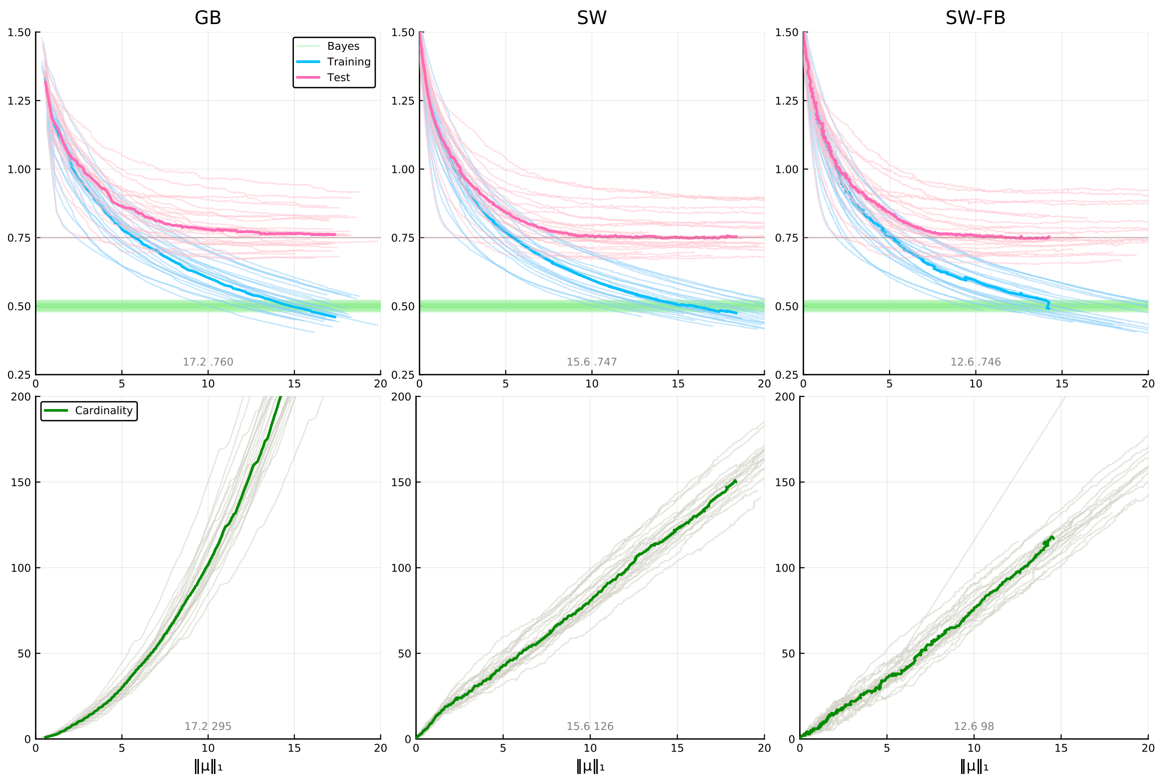

Figure 7.2: Top row: Training, Test and Bayes Errors for GB, SW and SW-FB. Bottom row: Increasing cardinality with $\|\boldsymbol{\mu}\|_1$. Both versions of SW achieve best test error of .7465; while the modified version seems to use fewer ReLUs. GB does marginally (5%) worse with a test error of .760 with 295 ReLUs. GB also adds new predictors at a quadratic rate with increasing $\|\boldsymbol{\mu}\|_1$.

Figure 7.2 compares the error rates and the sparsity of the three methods. It is clear that gradient boosting as popularly implemented hurts both performance and sparsity of the fit. It helps to get better fit by going slowly, i.e. not adding the next weak-learner in a greedy fashion. It also helps to revist existing weak-learners and increment them before trying to add a new one. The SW-FB method tries to solve the infinite dimensional lasso (4.6) exactly and as a consequence does better in terms of both generalization and sparsity than gradient boosting which is quick and dirty.

In the above context, adding a damping parameter γ to GB might help the generalization error, but it would severely increase the already very high cardinality by a factor of $1/\gamma$ which can be anywhere between 3-100 times. The same effect of better generalization can be achieved via the SW or SW-FB methods while reducing the cardinality!

7.3.3 Effect of Alternating Minimization

Now we study the effect of AD on the three methods FW, GB and SW: (giving us ADFW, ADGB, ADSW). Figure 7.3 compares the performance of the normal versions of the three algorithms with their AD counterparts. It is clear that the median test error improves by approximately four points across the board. This is a 13% improvement for Frank-Wolfe, 15% for Gradient Boosting and 19% for Stagewise (all relative to the Bayes error). We also see that in the case of SW, the increase in cardinality as the norm of the coefficient vector increases is much slower. So ADSW achieves very low error rate of .705 with just 36 ReLUs.

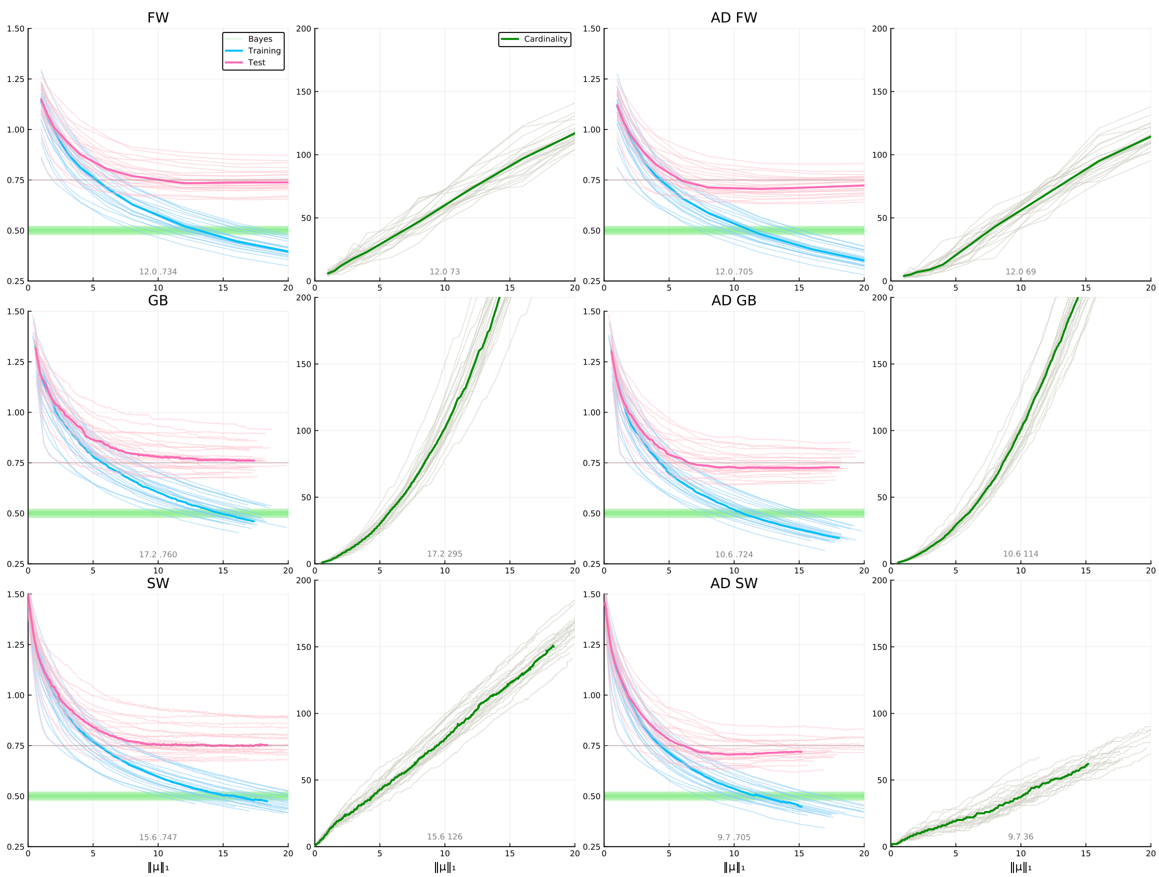

Figure 7.3: Errors Rates and Cardinalities for FW, GB and SW and their AD counter parts on the right. With AD, FW sees a performance improvement of 13%, GB 15% and SW 19% respectively. The best cardinality remains the same for FW. But it drops by half for GB and by nearly a fourth for SW!

We surmise that this efficiency comes from the fact that the lasso path tends to be very unstable when the covariates are highly correlated. In our case, as we have uncountably infinite predictors, with arbitrarily high correlations, this problem is manifest in the extreme. In Lasso with highly correlated predictors we see that as we progress on the path, a predictor is dropped and a relative is picked up, and this happens too often [Hastie et al., 2001]. Alternating Descent circumvents this problem by moving smoothly between a ω and its highly correlated neighbour without having to decrease one to zero and increase another. This avoids spurious atoms with low or negligible coefficient. This effect is further explored in Figure 7.4, from which it is obvious that with the AD option on, both SW and SW-FB end up being the same. This is once again because the necessity to drop predictors, as SW-FB does, is greatly diminished as they can be moved around in Ω. Although the equivalence of SW and SW-FB with AD is not always guaranteed, but can be expected to some degree.

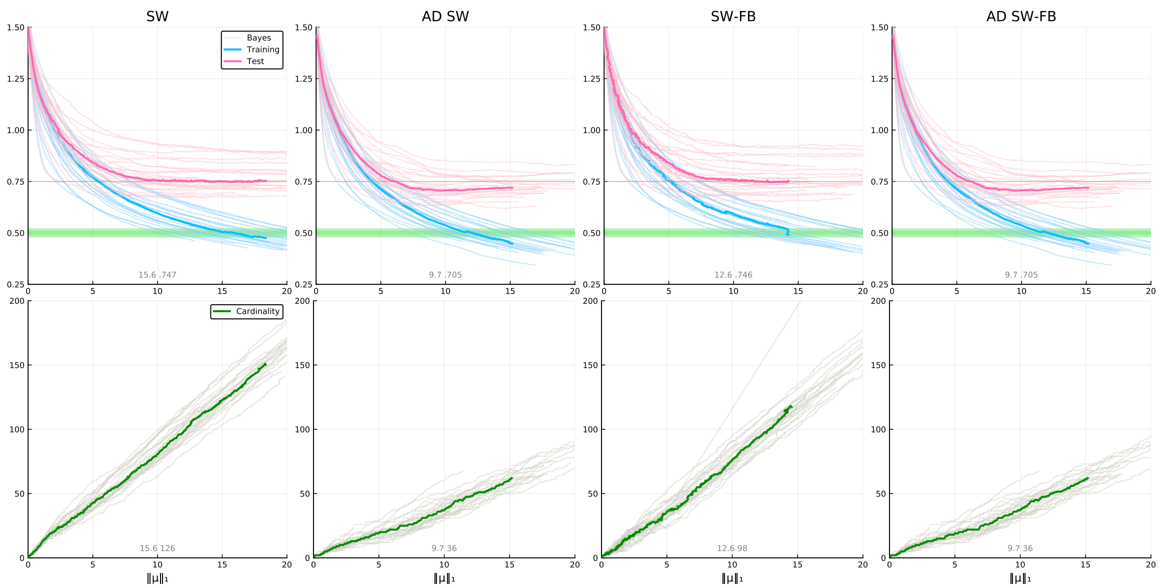

Figure 7.4: Comparison between SW and SW-FB without and with AD. AD improves performance by 16-19%. When using AD versions, both the algorithms end being almost the same. The cardinality in the AD versions also rises much more slowly. Overall ADSW seems like a very method for fitting our problem.

7.3.4 Rival Methods: NN, XGBT

In this section we compare the performance of our methods to that of popular rivals. As the problem we are trying to solve is a neural network, that will be one of the rivals. For a problem of such a nature, XGBoost seems to be the most popular method in the world, so we pick that. XGBoost uses trees as weak-learners. We use the default parameters for XGBoost. Note that the true function model that we are using is from [Friedman, 2001] — the Gradient Boosting with Trees paper. The model works well with trees with low order interactions, as each tree is picking only up to six variables. This is a much better approximation to the true model, as in contrast, our weak-learner the ReLU is not being told to exclude any variables. Despite this disadvantage and the maturity of XGBoost as a package, we see that our method still out-performs XGBoost on this simulation study. Both our method and XGBoost have been hyper-parameter tuned by the same amount to keep the comparison fair — we went with the default parameters for both. The default learning rate for XGB-Tree is 0.3, and we leave it at that. Reducing it might help generalization error, but adversely affects the cardinality and vice-versa. We also note that each tree of depth six, has more parameters than a simple ReLU that has only $p + 1$ parameters.

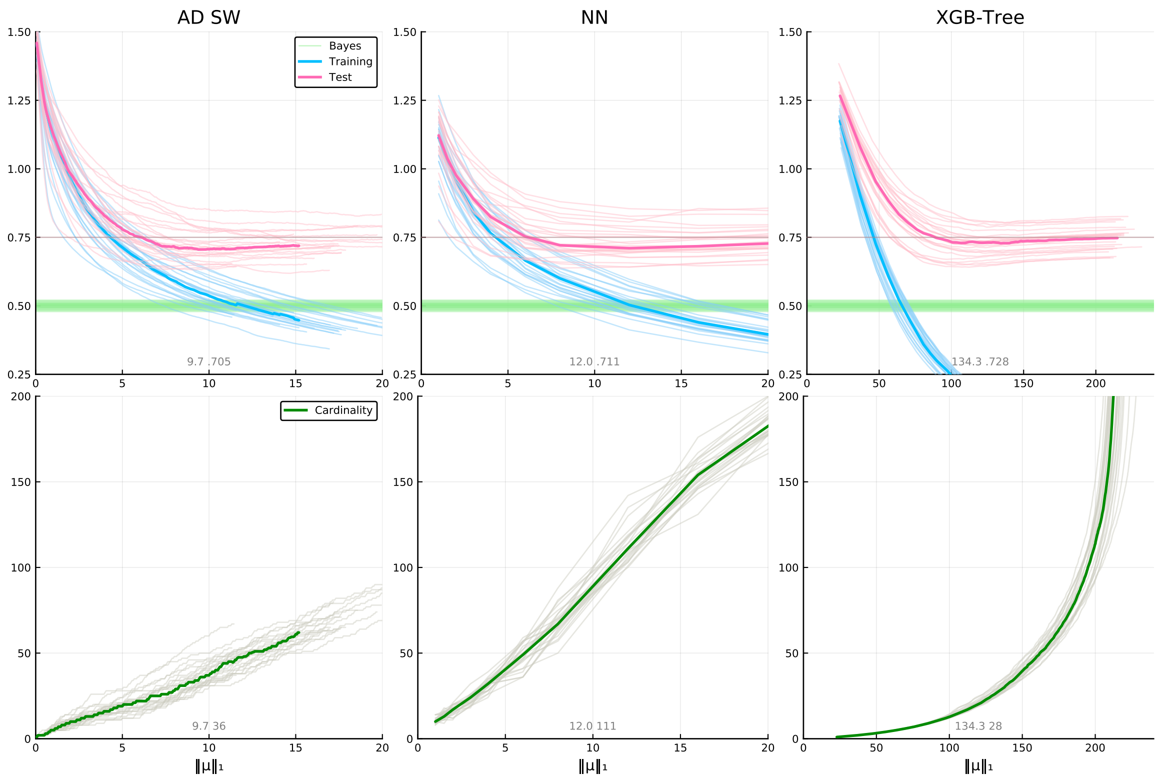

Figure 7.5: Comparison between ADSW, NN and XGBoost-Tree. We see that ADSW achieves a median test error of .705 with 36 ReLUs. NN .711 with 111 ReLUs. XGB-Tree .728 with 28 Trees. We see that ADSW improves by 11% over the XGBoost-Tree method. It also beats the highly tuned NN especially with regards to sparsity.

7.4 Classification Simulation

We now perform similar analysis for Logistic Regression, where the main measure of performance is the misclassification loss. The logistic loss is the convex surrogate we use as **L** in our model. The data is generated as in the case of linear regression in section 7.1, and as a final step it is rounded to 0 or 1. This results in a bayes misclassification rate of 19.1%. The results are shown in Table 7.1. Overall we see that ADSW gives the best or close to the best error rate with very few predictors (37 vs 213 for XGB-Tree). While XGB-Tree does give a 6% better performance than ADSW for this simulation, we note that XGB uses second order information about the loss, where as, all our methods use only first order information. This motivates us to examine second order methods to solver our problem (section 9.2). We also note that in our real data analysis presented in the next section, we see that the advantage of XGBoost is not apparent in the cross-validated setting, leading us to conclude that our methods are competitive to XGBoost and the 6% improvement is not across the board.

Table 7.1: Tabulated results for linear and logistic regressions. We report the best test loss achieved by each method and the corresponding cardinality. For logistic regression we report both the test deviance and the test misclassification rate. The bayes error rate for the regression problem is .5 and for the classification problem is 19.1%.

Algorithm	LinearRegression		LogisticRegression		
	MSE	Card.	Dev.	Miscln.%	Card.
FW	.734	73	.516	25.5	64
ADFW	**.705**	**69**	.503	**24.4**	**45**
GB	.760	295	.519	25.4	85
ADGB	.724	114	.505	24.7	81
SW	.747	126	.514	25.2	79
ADSW	**.705**	**36**	.500	**24.1**	**37**
SW-X	.746	98	.514	25.3	114
ADSW-X	.705	36	.503	24.2	33
NN	.711	111	.512	25.1	205
XGB-Tree	**.728**	**28**	.503	**23.8**	**213**

7.5 Sparsity Analysis

We now analyze the efficiency of the above methods in terms of frugality of atoms used. While it is popular belief that the more the weak-learners the better the ensemble, we would still want fewer weak-learners in the ensemble. This is because, if we are testing a lot of samples live in real-time the run-time of the fitting on these new test samples matters a lot. This is especially relevant in internet based services.

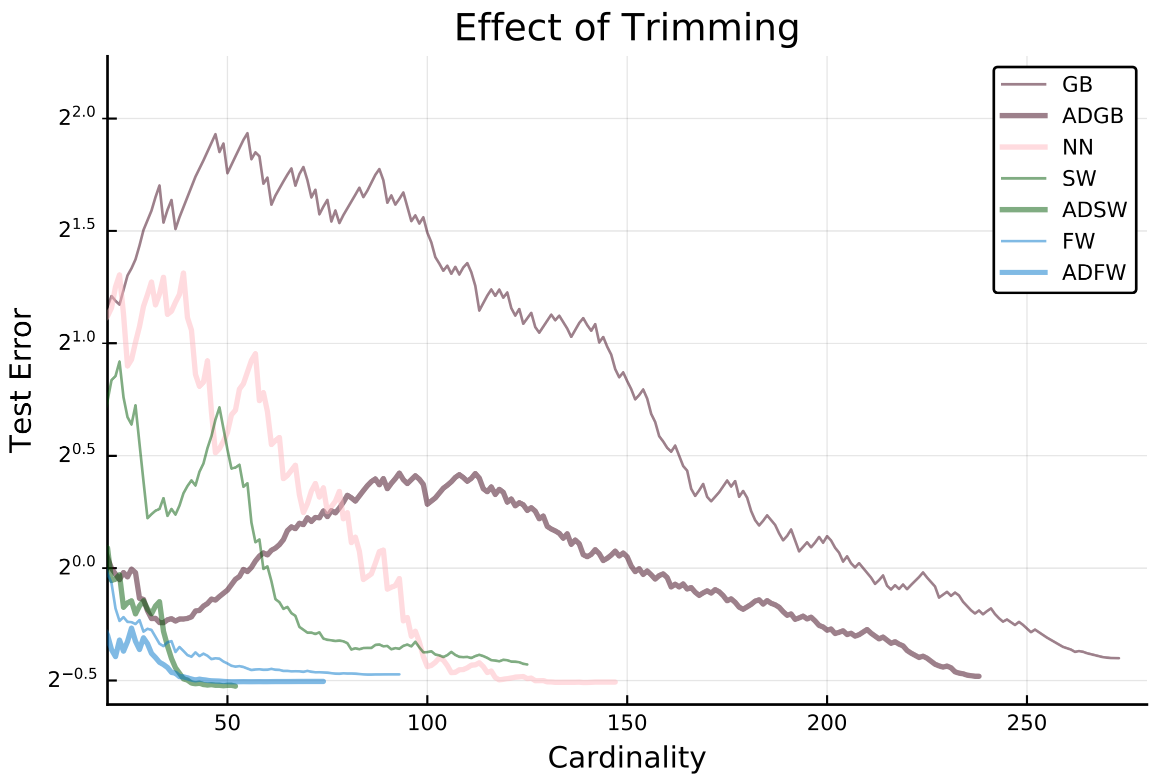

Figure 7.6: The effect of trimming the fit on Test Error. Error increases (from right to left) as we trim the fit by removing the nodes with the smallest coefficients. We see that ADSW is the most parsimonious of all the methods giving a test error below .7 with just 42 nodes. ADFW also compares well, although it performs the finite dimensional lasso at each step, giving it an advantage in that regard.

In popular belief, it is thought that the weak-learners tend to cancel themselves out and hence do not affect the fit. We instead want to see if that belief is even true. May be some of the methods do give very good predictors key to good prediction, where as some others give the same good ones with a bunch of unnecessary ones. In real world applications employing gradient boosting, it is common practice to use a post-processing step, where all the weak-learners are taken to be the new predictors and a lasso run on them to further trim the ensemble. This demonstrates the value of frugality of nodes. We want to study how much we can trim the fit while not losing predictive power. To this end we study the effect of removing the weakest of weak-learners from the ensemble. We take the best fit, corresponding to the lowest test error, for a given data and true function instance. Then we start trimming the network by removing the weakest atom. We set the lowest β_j to zero and check the test error. Then the next lowest, etc.

Figure 7.6 shows the effect of this trimming on the test error. We see that GB degrades very quickly when even the smallest weak-learners are removed from them model. This happens even though GB has a lot of predictors in it to begin with. This is because of the greediness of GB and we hypothesize that the later weak-learners are there just to compensate the former ones' mistakes, which are a by-product of the greediness. Removing these later smaller weak-learners exposes the over-fitting of the former ones, leading to bad generalization.

We see that ADSW wins again, it gets an error rate less than .7 with just 42 nodes. While even the NN deteriorates with less than 130 nodes. We see that ADFW is also very resilient to trimming, it seems to be the most resilient in the extreme case where we have just a handful of predictors. This behaviour is easily explained by the fact that FW refits the finite dimensional lasso after each iteration, thus leading to extra sparsity. This comes at a huge computational cost. Especially so in the case of ADFW, where the finite dimensional lasso is performed each time we perform the AD step, which means that we are running the finite dimensional lasso multiple times when a new predictor enters.

7.6 Real Data Study

In this section we study the performance of various methods on a real world classification problem. For this we use the Portugese Wine dataset. It has the following predictor variables: color, fixed_acidity, volatile_acidity, citric_acid, residual_sugar, chlorides, free_S02, total_S02, density, pH, sulphates and alcohol. Each of these twelve variables describes the various chemical properties of the wine. The entity we are trying to predict is the quality of wine as rated by humans. The quality of wine is ranked on a scale of one to ten. For the sake of our study, we classify the first five class as low and the rest as high quality respectively. We also remove some samples to make the problem more balanced. We normalize the data by applying log-transforms where necessary. While this is not necessary when using weak-learners that are invariant under monotone transformations (like trees), in the case of ReLUs it becomes important. This is analogous to the traditional Lasso or Ridge Regression setting where it is standard practice to normalize the predictors. In this real world example, we can not know the Bayes error rate. The irreducible error contributes equally to all the methods.

We apply the eight methods mentioned above, and compare them to a neural network and an XGBoost with trees. Model selection is done via nine fold cross-validation. We find the value of the norm of $\boldsymbol{\mu}$ that achieves the best cross-validated test error on the median of the nine folds, then we use that value on the entire training data and report the test error on the heldout data. The data has 5000 samples, of which 3600 are used for the cross-validated training, and 1400 samples are held out. We see that ADFW achieves the best error rate with just 95 ReLUs. We see that even

Table 7.2: Non-linear logistic regression on the modified wine dataset. $\|\boldsymbol{\mu}\|_1$ is picked by nine fold cross-validation. We see that ADFW, ADSW both perform well comparable to existing methods Neural Networks and XGBoost.

Algorithm	Cross-Validated			Heldout	
	Norm	Card.	Miscln.%	Card.	Miscln.%
FW	64.0	78	22.0	71	20.4
ADFW	100.	91	20.5	**95**	**17.9**
GB	34.9	786	21.8	800	19.0
ADGB	34.2	656	20.8	685	18.8
SW	56.5	182	20.5	193	19.0
ADSW	54.2	136	21.2	138	**19.2**
SWL	52.5	154	21.2	149	19.6
ADSWL	62.7	128	20.4	115	20.0
NN	14.7	300	22.2	300	19.4
XGB-Tree	-	89	17.8	79	**18.6**

on a real world example, some of our algorithms are competitive to XGBoost with trees, which is a time-honed well-tested method, regarded as the gold-standard for classification. We also point out that XGBoost with trees uses a second order approximation to the loss, while all our methods just use the gradient. In conclusion, this study extends the validity of our simulation study to the real world.

7.7 Timing Analysis

We now analyze the run time for various methods compared above. We report the run-times on the 25 problem instances referred to in the regression simulation study above. The box plot of results are shown in figure 7.7 and the median of the 25 runs are tabulated in Table 7.3.

XGBoost

We see that XG-Boost with trees has the fastest run-time with a median of seven seconds. This is to be expected since XGBoost is highly optimized for fitting time. Even finding the next weak-learner tree takes only $O(np \log n)$ time, since there is no multiplication involved in evaluating a tree.

Neural Network

As we would expect, the Neural network is at the other end of the spectrum, with the longest run time. Part of this long run-time is explained by the fact that we are not using any stochasticity

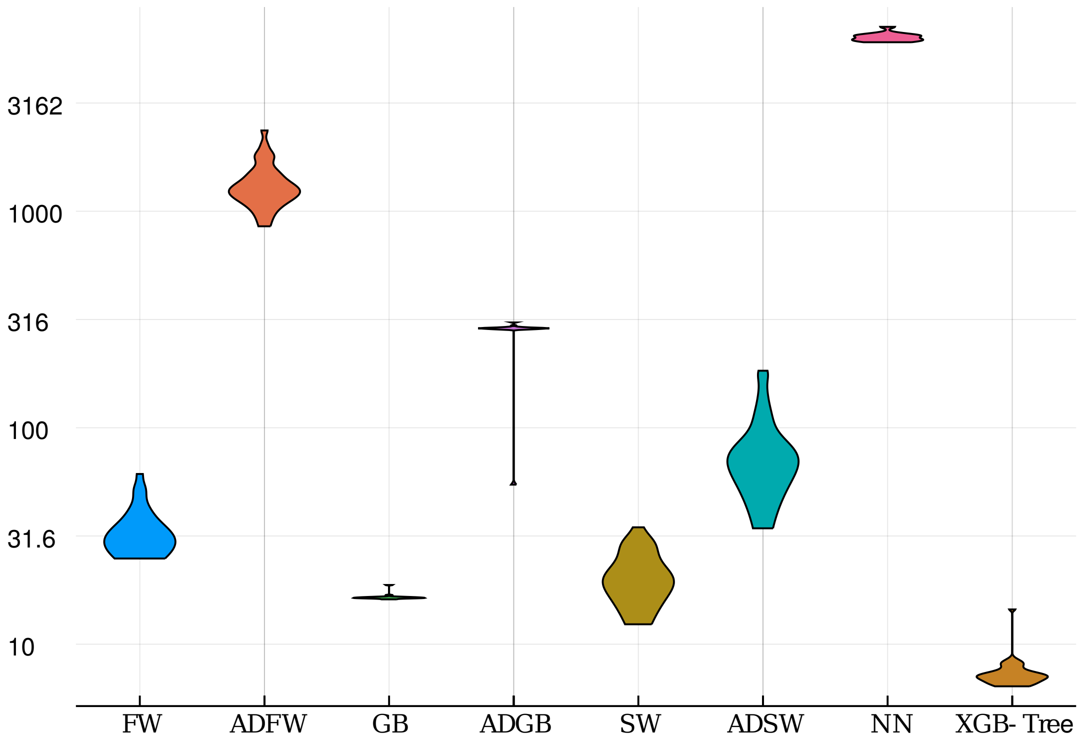

Figure 7.7: Run-times for the various algoritms. Methods using 'back-propagation' take a few orders of magnitude longer to run. Stochasticity will reduce the runtimes for them, but the relative positions should be the same.

Algorithm	Run-time
FW	32
ADFW	1286
GB	16
ADGB	288
SW	20
ADSW	74
NN	6365
XGB-Tree	7

Table 7.3: Median run time in seconds for each of the algorithms for the entire path.

in fitting the Neural Network. However, in comparison, we are not using stochasticity to obtain the next weak-learner or to perform alternating descent for any of the other methods either. The evaluation of the gradient and the jacobian, and the subsequent update step all involve matrix multiplications that are expensive. But this also applies to all the methods that use Alternating Descent in the ω space.

Gradient Boosting

This is expected to be fast like XGBoost is. There are two differences. One, the developer is different. XGBoost is written in C++ and is optimized for multi-threading. Two, GB with ReLUs does involve matrix multiplication, where as the costliest operation in XGBoost-Tree is sorting the values of x_{ij} along each j. Even then we see that GB is only twice as slow as XGB-Tree for a moderate sized problem with $n = 2000$.

Stagewise

Stagewise compares to GB and XGB-Tree with a median runtime of 20 seconds. It is expected to be slightly slower than GB, because GB uses the obtained weak-learner greedily as much as it can. In contrast, SW may or may not use the new predictor. In a lot of cases, the new predictor is just thrown out, leading to a lot of calls to the sub-routine **`best-`**ω. This discarding of predictors may also explain the high variance in the times for FW and SW as opposed to GB and XGB. But the slight increase in time from GB to SW is justified by better performance and better cardinality.

Frank-Wolfe

The runtime for FW with a median of 32s is comparable to that of GB(16s) and SW(20s). This is easily explained by the fact that all the three are calling the subroutine **`best-`**ω repeatedly. In addition, FW also refits the finite dimensional Lasso each time a new predictor is summoned. This explains the requirement of more time. One important thing to note is that, FW is a point algorithm, which is turned into a path algorithm using warm-starts from previous value of the bound on $\|\boldsymbol{\mu}\|_1$. The runtime for FW therefore depends on the discretization of the gird of norm-bounds. Here we fit over a grid of 11 values. This is rather arbitrary and we need more detailed study on how the run-time increases as we increase the fineness of the grid. This makes FW slightly less appealing compared to SW. Also FW needs a subroutine to fit the finite dimensional lasso *in bound form* which might not be readily available.

ADGB & ADSW

First we note that ADGB is significantly worse than ADSW by being almost four times slower to fit. Also note that the AD version of GB is slower than its vanilla counterpart by 18 times whereas the that of SW is slower only by four times. One possible explanation for this is that AD takes much more time to converge in the case of GB when the next predictor is added because we add a lot of it (as opposed to a small ϵ amount), this makes the jump in $\|\boldsymbol{\mu}\|_1$ rather huge and the active ω's need to be moved around a lot, since the main problem (4.6) changes a lot in terms of the bound on $\|\boldsymbol{\mu}\|_1$.

ADFW & ADSW

ADFW is way slower at $21\frac{1}{2}$ minutes than ADSW at $4\frac{4}{5}$ minutes. This is because the AD step for FW is much more complicated than that for SW. In ADFW, we iterate between moving ω's around and their coefficients around till convergence. This means that one AD step in ADFW is multiple AD steps for SW. Thus making it really slow. Understandably this makes ADFW more comparable to NN than to SW. This makes ADFW rather undesirable in comparison to ADSW.

In the end we can conclude from the timing and performance analysis that ADSW is the best overall procedure from all the presented methods. It has a runtime that is only ten times slower than XGBoost, while being generalizable to any class of non-linearities and by extension to sparse inverse problems. It is a 100 times faster than Neural Networks (relative to being ten times slower than XGBoost), while giving same or even better performance. It has virtually no tuning parameters, and is very quick to implement for any arbitrary loss.

Chapter 8

Framework

In the previous chapters we have seen seven algorithms to solve our problem. These algorithms are connected via the reuse of subroutines. In this section, we see how we can leverage these connections to efficiently build a framework that can be used to solve our infinite dimensional problem (4.6) for various choices of loss, activation and constraint sets using our algorithm of choice. This will also help us extend the scope of the work to any Sparse Inverse Problem even beyond Machine Learning.

8.1 Types, Subroutines, Algorithms

A problem is defined by the data $\mathbf{X}, \mathbf{y}$ that we are trying to model. We choose our model, the infinite dimensional convex neural network. This model asks us to pick our choice of loss **L** dictated by the nature of $\mathbf{y}$, then we are asked to pick an activation ϕ and a constraint set Ω. Once we have these, we need to implement the required subroutines for these types. For example the **`gradient`** subroutine depends only on the loss, and a different choice of ϕ does not affect the calculation of this gradient. Where as the **`jacobian`** that is used in alternating descent depends only on the activation and does not depend on the loss. The subproblem routine **`best-`**ω depends on the tightly coupled activation-constraint pair. These dependencies of the subroutines on the types are summarized in Table 8.1.

Once we have the subroutines implemented we can build the algorithms using these subroutines as the main building blocks along with some glue code to integrate a set of subroutines into an algorithm. For e.g., the FW algorithm needs the gradient of the loss, the next best predictor depending on (ϕ, Ω) and a subroutine to perform the finite dimensional lasso in bound form. It could optionally use back-propagation code to perform alternating descent. Similarly, the SW algorithm needs a different subset of subroutines. These dependencies between the algorithms and the subroutines are summarized in Table 8.2. Figure 8.1 combines these two tables to give a better idea of the connections between the 'Types', the 'Subroutines' and the 'Algorithms'.

Table 8.1: Subroutines and the Types they are specialized for. The Types define the problem and the subroutines form the building blocks for Algorithms. When introducing a new instance of a Type (like Poisson loss), the corresponding Subroutines need to be extended.

Types → Subroutines ↓	Loss	Ω	ϕ
gradient	✓		
jacobian			✓
backprop-ω	✓		✓
best-ω		✓	✓
lasso-on-$\mathcal{A}$	✓		
line-search	✓		
Examples	MSE, Logit, ...	$\Omega_2, \Omega_{\mathbf{X}}$, ...	ReLU, ReQU, ...

8.2 Implementation with Multiple Dispatch

Consider for example the Loss Type. It has multiple subtypes, like MSE, Logit, Poisson etc. Each of these losses need to have their gradient defined. This is presented as a detailed example in the next section. We use Julia where the loss can be defined as *methods* for the different types.

Methods in Julia

Our programming language of choice is Julia [Bezanson et al., 2017]. Julia is a high-level dynamic programming language designed for high-performance numerical analysis and computational science. It is a Type based language. For e.g., numbers can be of type Int32, Int64, Float32, Float64, etc. And functions have methods. Each method of a function is called for a different type. For e.g., one can define two methods for a function that checks if a number is round. We know all integers are

Table 8.2: Algorithms and the subroutines they employ. A '*' means that the algorithm uses **backprop-**ω when we choose to employ alternating descent (AD).

Algorithms → Subroutines ↓	NN	FW	SW	GB
gradient	✓	✓	✓	✓
jacobian	✓			
backprop-ω	✓	*	*	*
best-ω		✓	✓	✓
lasso-on-$\mathcal{A}$		✓		
line-search				✓

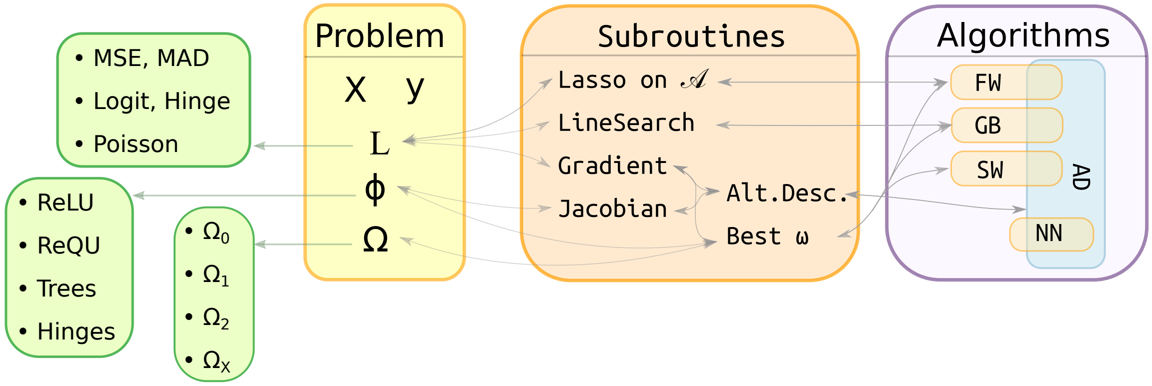

Figure 8.1: The relationship between the Types, the Subroutines and the Algorithms.

round numbers, so we can return true based on the type Integer, else we check to see if the number is really round.

```
isround(x::Integer) = false
isround(x::AbstractFloat) = (x==round(x))
```

Similarly in our case we can define the gradient based on the loss subtype as:

```
gradient(l::MSE, z, y) = z - y
gradient(l::Logit, z, y) = sigmoid(z) - y
gradient(l::Poisson, z, y) = exp(z) - y
gradient(l::Any, z, y) = error("Not implemented.")
```

At compile time the type of the loss is known and julia can substitute the code from the correct method into the algorithm that is being compiled (say FW) to generate efficient code without 'switch' clauses. Additionally, the Algorithm need not be changed when we are adding a new loss type. Although this single argument dispatch system is available in most languages (including all objected oriented languages), Julia allows for multiple dispatch based on the types of *all* the arguments. This is helpful in implementing the subroutine to get the next best predictor based on the types of both activation and the constraint set.

```
function nextbestpredictor(activation::ReLU, Omg::Omega2)
      # vanilla frank-wolfe based solver
end
function nextbestpredictor(activation::Any, Omg::OmegaX)
      # non-linear least squares solver
      ...
      J = jacobian(activation)
      ...
```

```
    omega = project(Omg, omega)
    ...
end
```

Here again the correct method for the **jacobian** is compiled in based on the type of the activation. Similarly the correct method for the projection onto Ω is compiled based on the type of Ω.

This Type System and Multiple Dispatch based framework helps us abstract the algorithms and decouple them from the implementation details of the subroutines (which can be very very complicated, like for **best-**ω). It simultaneously allows us to generate efficient and specialized code to handle any combination of loss, activation and constraint at runtime. This allows a new user of our package to import it, define new types and subroutines in their code (for say new losses or activations) and use any of our seven algorithms without modifying our package. In the end, they get compiled code that is well optimized.

For e.g. any Sparse Inverse Problem can be solved using any of our seven algorithms by defining the necessary subroutines. Say you want to use SW, all it needs is that you define the **gradient** of the loss, the **best-**ω based on the activation-constraint set pair, and the **jacobian** based on the activation and you can use SW and ADSW to fit your problem.

8.3 Loss and Gradient

Previously we have seen examples of how the Jacobian $\nabla_\omega \phi(\mathbf{X};\omega)$ can be calculated for the ReLU activation function. In that vein we now present a few losses and their gradients as an appetizer. This illustrates how a typical function — the **evaluation** and **gradient** of a loss type — is specified via the methods for each type. This section also illustrates why the gradient is also called the *negative generalized residual*.

We need a function to evaluate the loss $\mathbf{L}\left(\mathbf{z};\mathbf{y}\right)$ given a fit $\mathbf{z}$ and target $\mathbf{y}$. This is straigt forward, like

$$
\begin{aligned}
\mathbf{L}_{\texttt{MSE}}(\mathbf{z};\mathbf{y}) &= \tfrac{1}{2}\left\|\mathbf{z}-\mathbf{y}\right\|_2^2 \\
\mathbf{L}_{\texttt{MAD}}(\mathbf{z};\mathbf{y}) &= \left\|\mathbf{z}-\mathbf{y}\right\|_1 \\
\mathbf{L}_{\texttt{Logit}}(\mathbf{z};\mathbf{y}) &= -\mathbf{z}\cdot\mathbf{y} + \Sigma\log(1+\exp\left(\mathbf{z}\right)) \\
\mathbf{L}_{\texttt{Poisson}}(\mathbf{z};\mathbf{y}) &= -\mathbf{z}\cdot\mathbf{y} + \Sigma\exp\left(\mathbf{z}\right)
\end{aligned}
$$

We also need the gradients w.r.to $\mathbf{z}$ which are readily calculated from the above expressions as:

$$\begin{aligned}
\nabla_{\mathbf{z}}\mathbf{L}_{\texttt{MSE}}(\mathbf{z};\mathbf{y}) &= \mathbf{z}-\mathbf{y} && = \hat{\mathbf{y}}-\mathbf{y} \\
\nabla_{\mathbf{z}}\mathbf{L}_{\texttt{MAD}}(\mathbf{z};\mathbf{y}) &= \operatorname{sign}(\mathbf{z}-\mathbf{y}) && \\
\nabla_{\mathbf{z}}\mathbf{L}_{\texttt{Logit}}(\mathbf{z};\mathbf{y}) &= \operatorname{sigmoid}(\mathbf{z})-\mathbf{y} && = \hat{\mathbf{y}}-\mathbf{y} \\
\nabla_{\mathbf{z}}\mathbf{L}_{\texttt{Poisson}}(\mathbf{z};\mathbf{y}) &= \exp(\mathbf{z})-\mathbf{y} && = \hat{\mathbf{y}}-\mathbf{y}
\end{aligned}$$

It is now obvious why we call the gradient $\nabla_{\mathbf{z}}\mathbf{L}$ as the negative generalized residual. For GLMs with the canonical link, it corresponds to the negative of the residual (once the $\mathbf{z}$ are transformed to the domain of $\mathbf{y}$ by applying the mean function).

Chapter 9

Second Order Solvers

So far we have seen various first order methods to solve our problem (4.6). These methods linearize the objective at the current estimate and minimize the linearized objective to hit a vertex which is a single predictor. This predictor is added to the active set, and the weights for all predictors in the active set are updated by rules that differ from algorithm to algorithm. In first order methods, the gradient of the loss with respect to the fit plays a major role. Now we will consider methods that go beyond the linearlize and minimize paradigm and consider second derivative of the loss with respect to the fit. Given that the fit is linear in terms of the measure, this is all the second order information we need.

We start our discussion with the LARs method. LARs applies only to Least Squares problems, where second order information is all we need. That is LARs tackles the LS problem head on, and solves the problem exactly (in finite dimensions) up to machine precision. Then we will try to apply second order methods to general losses beyond the quadratic. We present what we call HessianBoost (a generalization of XGBoost) and Infinite Dimensional Coordinate Descent. The former works on the bound form of the problem (4.6), where as the latter works on its lagrangian form (9.30). We do not present any simulation studies for these two general second order methods.

9.1 Infinite Dimensional LARS

We now revisit our main problem (4.6) specifically for the special case of $\mathbf{L}$ being the Mean-Squared Error Loss (MSE). In this case we can perform LARS [Efron et al., 2004] in its infinite dimensional version [Rosset et al., 2007]. While [Rosset et al., 2007] present the infinite dimensional LARs, they do not apply or implement it to a truly infinite case, their demonstrative example is the Trend Filtering problem [Kim et al., 2009], but we know that it reduces to a finite dimensional Lasso problem [Tibshirani, 2014]. We now present the infinite dimensional LARs and detail it as it applies to the ReLU activation function. We start with a presentation of the finite dimensional LARs algorithm.

9.1.1 Traditional LARs

We want solve the finite dimensional problem

$$\operatorname*{argmin}_{\boldsymbol{\beta}} \tfrac{1}{2} \left\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\right\|_2^2 \quad \text{s.t.} \quad \left\|\boldsymbol{\beta}\right\|_1 \leqslant C. \tag{9.1}$$

Here $\mathbf{X} \in \mathbb{R}^{n \times p}$. For simplicity of exposition we will the Negation Closure Assumption 1 holds. i.e. $\mathbf{X}$ contains the negation of all its columns. This makes the solution $\boldsymbol{\beta}$ unidentifiable. So we use the negation closed reformulation (4.17) for the finite-dimensional case.

$$\operatorname*{argmin}_{\boldsymbol{\beta}} \tfrac{1}{2} \left\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\right\|_2^2 \quad \text{s.t.} \quad \left\|\boldsymbol{\beta}\right\|_1 \leqslant C, \quad \boldsymbol{\beta} \succ 0, \tag{9.2}$$

or its lagrangian equivalent

$$\boldsymbol{\beta}(\lambda) = \operatorname*{argmin}_{\boldsymbol{\beta}} \tfrac{1}{2} \left\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\right\|_2^2 + \lambda \left\|\boldsymbol{\beta}\right\|_1 \quad \text{s.t.} \quad \boldsymbol{\beta} \succ 0. \tag{9.3}$$

We do this only for the ease of exposition. As we noted earlier this does not alter the problem in any way.

Given that the solution path for the above problem for a sequence of λs is piecewise linear, LARs builds the path for $\boldsymbol{\beta}(\lambda)$ in knots of λ_k. λ is also called the 'dual variable' (in the sense of the generalized lasso [Tibshirani et al., 2011]. From the Lasso KKT conditions, λ is also the maximal covariance of any of the predictors $\mathbf{X}_j$ with the residuals $\mathbf{r}(\lambda)$ at any given point on the path; and only variables that have maximal covariance with the $\mathbf{r}$ can have a non-zero coefficient. This set is the active set

$$\mathcal{A}(\lambda) = \{j \mid \beta_j(\lambda) \neq 0\} \subset \{j \mid \langle X_j, \mathbf{r}(\lambda)\rangle = \lambda\}.$$

where

$$\mathbf{r}(\lambda) = \mathbf{y} - \mathbf{X}\boldsymbol{\beta}(\lambda)$$

The LARs piecewise linear coefficient path changes slope at the knots. Subscript k is used to denote the k^{th} knot. Thus $\mathbf{r}_k, \beta_k$, etc denote values at that knot. For values at any point on the path, we will use the notation $\mathbf{r}(\lambda), \boldsymbol{\beta}(\lambda)$, etc. to denote the residuals, coefficients, etc. At the knots we simply notate $\mathbf{r}_k := \mathbf{r}(\lambda_k)$, $\mathcal{A}_k := \mathcal{A}(\lambda_k)$, etc.

At each knot, we augment the coefficients $\boldsymbol{\beta}(\lambda)$ linearly in the dual-variable λ,

$$\boldsymbol{\beta}(\lambda) = \boldsymbol{\beta}(\lambda_k) + (\lambda_k - \lambda)\,\boldsymbol{\delta}(\lambda_k) \quad \text{for} \quad \lambda_k > \lambda \geqslant \lambda_{k+1}.$$

So the fit $\mathbf{z}(\lambda) = \mathbf{X}\boldsymbol{\beta}(\lambda)$ is also augmented linearly in the dual-variable λ,

$$\mathbf{z}(\lambda) = \mathbf{z}(\lambda_k) + (\lambda_k - \lambda)\,\boldsymbol{\eta}(\lambda_k) \quad \text{for} \quad \lambda_k > \lambda \geqslant \lambda_{k+1},$$

and so is $\mathbf{r}(\lambda) = \mathbf{y} - \mathbf{z}(\lambda)$

$$\mathbf{r}(\lambda) = \mathbf{r}(\lambda_k) - (\lambda_k - \lambda)\,\boldsymbol{\eta}(\lambda_k) \quad \text{for} \quad \lambda_k > \lambda \geqslant \lambda_{k+1}.$$

Here $\boldsymbol{\delta}_k := \boldsymbol{\delta}(\lambda_k)$ is the direction in which $\boldsymbol{\beta}(\lambda)$ moves between the knots, and

$$\boldsymbol{\eta}_k := \boldsymbol{\eta}(\lambda_k) = \mathbf{X}\boldsymbol{\delta}(\lambda_k) \tag{9.4}$$

is the direction that makes equal angle (equal covariance) with all the members of the active set $\mathcal{A}_k$ — so that as we decrease λ, the covariance of the members of the active set to the residual are tied to be equal (until the active set itself changes). We reach the next knot when one of the inactive predictors matches the maximal covariance with residuals that the active set enjoys. [1]

$$(\lambda_{k+1}, j_{k+1}) = \max_{\lambda<\lambda_k}, \operatorname*{argmax}_{j\notin\mathcal{A}_k} \{(\lambda, j) \mid \langle \mathbf{r}(\lambda), \mathbf{X}_j\rangle = \langle \mathbf{r}(\lambda), \mathbf{X}_{\mathcal{A}_k}\rangle\} \tag{9.5}$$

Then it is added to the active set and $\boldsymbol{\eta}$ is updated. Note that the more the fit is augmented (with $\boldsymbol{\eta}$), the more its covariance with the active set degrades, making it possible for inactive predictors to catch-up: for $\lambda_k > \lambda \geqslant \lambda_{k+1}$,

$$\begin{aligned}\langle \mathbf{r}(\lambda), \mathbf{X}_{\mathcal{A}_k}\rangle &= \langle \mathbf{r}_k, \mathbf{X}_{\mathcal{A}_k}\rangle - (\lambda - \lambda_k)\,\langle \boldsymbol{\eta}, \mathbf{X}_{\mathcal{A}_k}\rangle \\ &= \lambda_k - (\lambda - \lambda_k)\,\langle \boldsymbol{\eta}, \mathbf{X}_{\mathcal{A}_k}\rangle\end{aligned}$$

We initialize the iteration count k, and the corresponding fit and coefficients $\mathbf{z}_0, \boldsymbol{\beta}_0$ to zero. The active set is initialized to the the variable with maximum covariance with $\mathbf{r}_0 := \mathbf{y}$, and the above iterative process is repeated until none of the variables are correlated to the residuals, i.e. $\lambda = 0$. Figure 9.1 shows a plot of the covariances of the predictors as a function of λ for a diabetes dataset. We present the finite dimensional LARs in full detail in Algorithm 7 and then move onto the infinite dimensional version.

9.1.2 Infinite Dimensional LARs

Let us now consider the infinite dimensional version. Everything remains the same, except that the search in (9.7) for (the index of) the next predictor entering the model is over Ω as opposed to

[1] Here we abuse notation slightly. As $\langle \mathbf{r}(\lambda), \mathbf{X}_{\mathcal{A}_k}\rangle$ is a constant vector, it is treated as a constant in equalities with scalars. Similarly $\langle \boldsymbol{\eta}, \mathbf{X}_{\mathcal{A}_k}\rangle$.

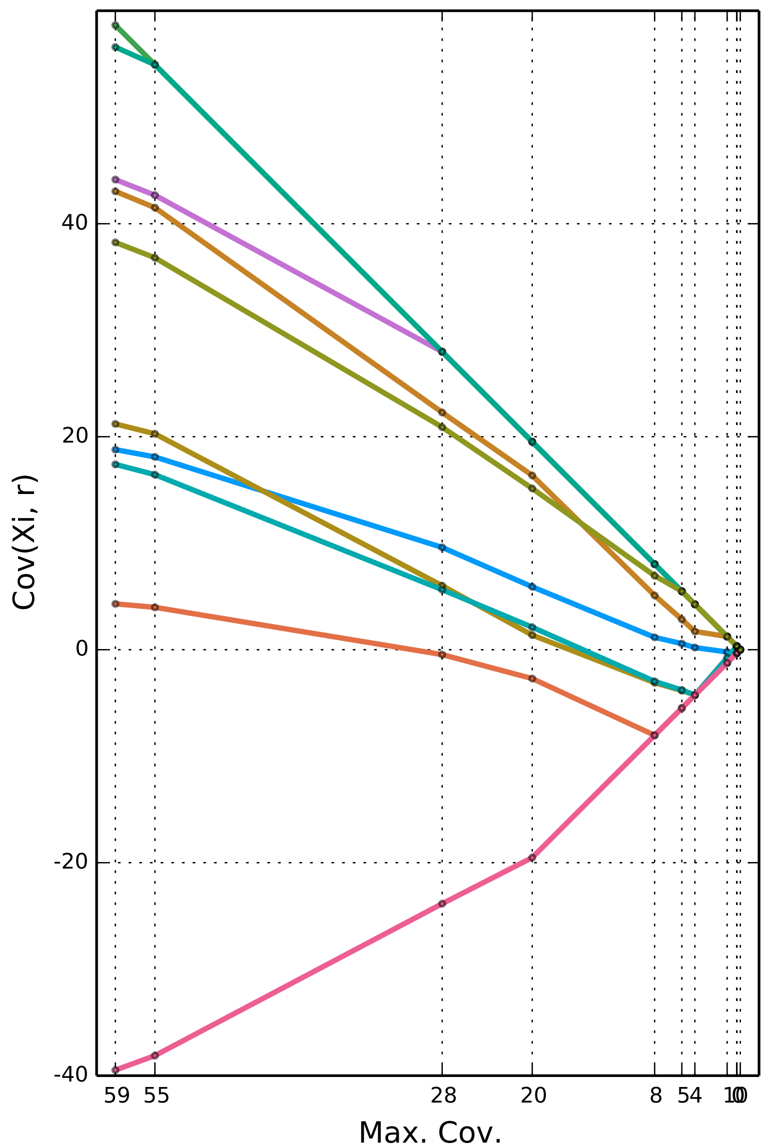

Figure 9.1: LARs example for the diabetes data without negation closure. The active set accepts newer predictors as their covariance with the residual becomes maximal (in absolute value, as we do not augment $\mathbf{X}$ with negations $\pm\mathbf{X}_j$). The goal is to find the integer-index of the next predictor that attains maximal-covariance. In the ∞-dim case, we need to find the real-valued-index of the next predictor-function that attains maximal-covariance.

Algorithm 7 Finite Dimensional LARs

Input: $\mathbf{X}, \mathbf{y}$
Initialize:

- Knot $k = 0$
- Coefficients $\boldsymbol{\beta}_0 = 0$
- Fit $\mathbf{z}_0 = 0$
- Residuals $\mathbf{r}_0 = \mathbf{y} - \mathbf{z}_0 = \mathbf{y}$
- Maximal covariance $(j^*, \lambda_0) = \text{argmax}, \max_j \langle \mathbf{y}, \mathbf{X}_j \rangle$
- Active set $\mathcal{A}_0 = \{j^*\}$
- Equi-angular unit vector $\boldsymbol{\eta}_0 = \mathbf{X}_{j^*} / \|\mathbf{X}_{j^*}\|_2$
- Direction of coefficients $\boldsymbol{\delta}_0$ such that $\boldsymbol{\eta}_0 = \mathbf{X}\boldsymbol{\delta}_0$, i.e. $\boldsymbol{\delta}_0 = e_{j^*} / \|\mathbf{X}_{j^*}\|_2$
- For $\lambda_k \geqslant \lambda \geqslant \lambda_{k+1}$, the fit and the residuals are
  - $\mathbf{z}(\lambda) = \mathbf{z}_k + (\lambda_k - \lambda)\boldsymbol{\eta}_k$
  - $\mathbf{r}(\lambda) = \mathbf{r}_k - (\lambda_k - \lambda)\boldsymbol{\eta}_k = \mathbf{y} - \mathbf{z}(\lambda)$

repeat
$k \leftarrow k + 1$
Find the next variable to achieve maximal covariance with the current residual $\mathbf{r}(\lambda)$

$$\lambda_k, j_k = \max_{\lambda < \lambda_{k-1}}, \underset{j \notin \mathcal{A}_{k-1}}{\text{argmax}} \left\{ \lambda, j \;\middle|\; \langle \mathbf{r}(\lambda), \mathbf{X}_j \rangle = \left\langle \mathbf{r}(\lambda), \mathbf{X}_{\mathcal{A}_{k-1}} \right\rangle \right\} \tag{9.6}$$

Update

- $\boldsymbol{\beta}_k = \boldsymbol{\beta}_{k-1} + (\lambda_{k-1} - \lambda_k)\boldsymbol{\delta}_{k-1}$
- $\mathbf{z}_k = \mathbf{z}_{k-1} + (\lambda_{k-1} - \lambda_k)\boldsymbol{\eta}_{k-1}$
- $\mathbf{r}_k = \mathbf{y} - \mathbf{z}_k$
- $\mathcal{A}_k = \mathcal{A}_{k-1} \cup \{j_k\}$
- $\boldsymbol{\eta}_k$ as unit-vector equi-angular to $\mathbf{X}_{\mathcal{A}_k}$, i.e. $\mathbf{X}_{\mathcal{A}_k}^T \boldsymbol{\eta}_k = c\mathbb{I}(\mathcal{A}_k)$
- $\boldsymbol{\delta}_k$ such that $\boldsymbol{\eta}_k = \mathbf{X}\boldsymbol{\delta}_k$

until $\mathcal{A}_k = [p]$ or $\lambda_k = 0$
Return: $\{\boldsymbol{\beta}_1, \cdots, \boldsymbol{\beta}_k\}$

$\{1, \cdots, p\}$.

At step k we have the atomic coefficient measure, $\boldsymbol{\mu}_k$; the active set, $\mathcal{A}_k = \text{support}(\boldsymbol{\mu}_k)$; the current maximal covariance, λ_k; the equi-angular unit vector, $\boldsymbol{\eta}_k$; the residuals

$$\mathbf{r}(\lambda) = \mathbf{r}_k - (\lambda - \lambda_k)\,\boldsymbol{\eta}_k \quad \text{for} \quad \lambda < \lambda_k.$$

Analogous to (9.5) in the finite dimensional case, we need to find which predictor enters next and when, [2]

$$(\lambda_{k+1}, \omega_{k+1}) = \max_{\lambda < \lambda_k}, \operatorname*{argmax}_{\omega \in \Omega \setminus \mathcal{A}_k} \{\lambda, \omega \mid \langle \phi(\mathbf{X}; \omega), \mathbf{r}(\lambda) \rangle = \langle \phi(\mathbf{X}; \mathcal{A}_k), \mathbf{r}(\lambda) \rangle\}. \tag{9.7}$$

We then update $\mathcal{A}$, $\boldsymbol{\mu}$, $\boldsymbol{\eta}$, $\mathbf{r}$ etc. like we do in the finite dimensional case. The main question here is how to solve (9.7) efficiently to find ω_{k+1} from amongst uncountably many. We will tackle this hard subproblem now.

9.1.3 LARs Subproblem

In the special case of MSE loss, we can perform LARs in infinite dimensions. The NP-hard subproblem of finding the next predictor is given by (9.7).

Revisiting traditional LARs, consider the simple Lasso problem of $\mathbf{y} \sim \mathbf{X}$. At a point on the path (between the first two knots (wlog)), we have the equi-angular direction of increasing fit $\boldsymbol{\eta}$. As we add $\gamma \equiv (\lambda_{\mathtt{max}} - \lambda)$ amount of $\boldsymbol{\eta}$ to the fit, the covariance of the j^{th} variable with the residuals is

$$\begin{aligned} c_j(\gamma) &= \langle X_j, \mathbf{r}(\gamma) \rangle \\ &= \langle X_j, \mathbf{y} - \gamma \boldsymbol{\eta} \rangle \\ &= \langle X_j, \mathbf{y} \rangle - \gamma \langle X_j, \boldsymbol{\eta} \rangle \\ &:= c_j - \gamma a_j. \end{aligned} \tag{9.9}$$

That is, the covariance decreases linearly in γ. Similarly for any of the maximally covarying predictors already in the the active set $\mathcal{A}$,

$$\begin{aligned} C(\gamma) &= \langle \mathbf{X}_\mathcal{A}, \mathbf{r}(\gamma) \rangle \\ &= \langle \mathbf{X}_\mathcal{A}, \mathbf{y} - \gamma \boldsymbol{\eta} \rangle \\ &= \langle \mathbf{X}_\mathcal{A}, \mathbf{y} \rangle - \gamma \langle \mathbf{X}_\mathcal{A}, \boldsymbol{\eta} \rangle \\ &:= C - \gamma A. \end{aligned} \tag{9.10}$$

A predictor $\mathbf{X}_j$ catches up (in covariance with the residuals) with the active set when we add γ_j

[2] Here we slightly abuse notation. As $\langle \phi(\mathbf{X}; \mathcal{A}_k), \mathbf{r}(\lambda) \rangle$ is a constant vector, it is treated as a constant in equalities with scalars. Similarly $\langle \phi(\mathbf{X}; \mathcal{A}_k), \boldsymbol{\eta} \rangle$.

Algorithm 8 Infinite Dimensional LARs

Input: $\mathbf{X}, \mathbf{y}$
Choices: ϕ, Ω
Initialize:

- Knot $k = 0$
- Fit $\mathbf{z}_0 = 0$
- Residuals $\mathbf{r}_0 = \mathbf{y} - \mathbf{z}_0 = \mathbf{y}$
- Maximal covariance $(\omega_0, \lambda_0) = \text{argmax}, \max_{\omega \in \Omega} \langle \mathbf{y}, \phi(\mathbf{X}; \omega) \rangle$
- Coefficient measure $\boldsymbol{\mu}_0 = (\boldsymbol{\beta}_0 = [0], \mathcal{A}_0 = \{\omega_0\})$
- Equi-angular unit vector $\boldsymbol{\eta}_0 = \phi(\mathbf{X}; \omega_0) / \|\phi(\mathbf{X}; \omega_0)\|_2$
- Direction of coefficients $\boldsymbol{\delta}_0$ such that $\boldsymbol{\eta}_0 = \phi(\mathbf{X}; \mathcal{A}_0)\boldsymbol{\delta}_0$, i.e. $\boldsymbol{\delta}_0 = [1/\|\phi(\mathbf{X}; \mathcal{A}_0)\|_2]$
- For $\lambda_k \geqslant \lambda \geqslant \lambda_{k+1}$, the fit and the residuals are
  - $\mathbf{z}(\lambda) = \mathbf{z}_k + (\lambda_k - \lambda)\boldsymbol{\eta}_k$
  - $\mathbf{r}(\lambda) = \mathbf{r}_k - (\lambda_k - \lambda)\boldsymbol{\eta}_k = \mathbf{y} - \mathbf{z}(\lambda)$

repeat
$k \leftarrow k + 1$
Find the next predictor to achieve maximal covariance with the current residual $\mathbf{r}(\lambda)$

$$(\lambda_k, \omega_k) = \max_{\lambda < \lambda_{k-1}}, \operatorname*{argmax}_{\omega \in \Omega / \mathcal{A}_{k-1}} \lambda \quad \text{s.t.} \quad \langle \mathbf{r}(\lambda), \phi(\mathbf{X}; \omega) \rangle = \langle \mathbf{r}(\lambda), \phi(\mathbf{X}; \mathcal{A}_{k-1}) \rangle \tag{9.8}$$

Update

- $\boldsymbol{\beta}_k = [\boldsymbol{\beta}_{k-1} + (\lambda_{k-1} - \lambda_k)\boldsymbol{\delta}_{k-1} \qquad 0]$
- $\mathbf{z}_k = \mathbf{z}_{k-1} + (\lambda_{k-1} - \lambda_k)\boldsymbol{\eta}_{k-1}$
- $\mathbf{r}_k = \mathbf{y} - \mathbf{z}_k$
- $\mathcal{A}_k = \mathcal{A}_{k-1} \cup \{\omega_k\}$
- $\boldsymbol{\eta}_k$ as unit-vector equi-angular to $\phi(\mathbf{X}; \mathcal{A}_k)$
- $\boldsymbol{\delta}_k$ such that $\boldsymbol{\eta}_k = \phi(\mathbf{X}; \mathcal{A}_k)\boldsymbol{\delta}_k$

until convergence
Return: Series of fits at each knot $\{\boldsymbol{\mu}_i = (\boldsymbol{\beta}_i, \mathcal{A}_i)\}_{i=1}^{k}$

---

amount of $\boldsymbol{\eta}$ to the fit such that

$$c_j(\gamma_j) = C(\gamma_j) \tag{9.11}$$
$$\implies c_j - \gamma_j a_j = C - \gamma_j A \tag{9.12}$$
$$\implies \gamma_j = \frac{C - c_j}{A - a_j}. \tag{9.13}$$

We need to find the predictor index j^* that has the least $\gamma_j \geqslant 0$:

$$j^* = \operatorname*{argmin}_j \frac{C - c_j}{A - a_j} \quad \text{s.t.} \quad \frac{C - c_j}{A - a_j} \geqslant 0. \tag{9.14}$$

By definition, C is the maximal covariance any of the predictors has to the residuals at any point on the path. This is achieved by the active set, hence for any $j \notin \mathcal{A}$, $c_j < C$, rendering the numerator positive. To satisfy $\gamma_j > 0$, we need to make sure the denominator is also positive, i.e. we minimize over the set $a_j < A$.

$$j^* = \operatorname*{argmin}_j \frac{C - c_j}{A - a_j} \quad \text{s.t.} \quad a_j \leqslant A. \tag{9.15}$$

Let us now extend it to the case of infinite predictors indexed by ω. We want to find the predictor that comes in next:

$$\begin{aligned} \omega^* &= \operatorname*{arginf}_{\omega \in \Omega} \gamma(\omega) \quad \text{s.t.} \quad \gamma(\omega) > 0 \\ &= \operatorname*{arginf}_{\omega \in \Omega} \frac{C - c(\omega)}{A - a(\omega)} \quad \text{s.t.} \quad a(\omega) \leqslant A. \end{aligned} \tag{9.16}$$

where,

$$\begin{aligned} c(\omega) &= \langle \mathbf{r}, \phi(X;\omega) \rangle \\ a(\omega) &= \langle \boldsymbol{\eta}, \phi(X;\omega) \rangle \end{aligned}$$

That is,

$$\omega^* = \operatorname*{arginf}_{\omega \in \Omega} \frac{C - \langle \mathbf{r}, \phi(X;\omega) \rangle}{A - \langle \boldsymbol{\eta}, \phi(X;\omega) \rangle} \quad \text{s.t.} \quad \langle \boldsymbol{\eta}, \phi(X;\omega) \rangle \leqslant A. \tag{9.17}$$

Quasi-convexity

If the activation function ϕ were just the multiplication function, $\phi(\mathbf{X};\omega) = \mathbf{X}\omega$, where $\omega \in \Omega \subset \mathbb{R}^p$ then we would have,

$$\omega^* = \operatorname*{arginf}_{\omega \in \Omega} \frac{C - \mathbf{r}^T X\omega}{A - \boldsymbol{\eta}^T X\omega} \quad \text{s.t.} \quad \boldsymbol{\eta}^T X\omega < A \tag{9.18}$$

This is a linear fractional function. Which is a quasi-convex function, with the quasi-gradient:

$$\frac{\boldsymbol{\eta}^T \mathbf{X}}{A - \boldsymbol{\eta}^T \mathbf{X}\omega} - \frac{\mathbf{r}^T \mathbf{X}}{C - \mathbf{r}^T \mathbf{X}\omega}. \tag{9.19}$$

So the problem we want to minimize is a quasi-convex function of $\phi(\mathbf{X};\omega)$. Which is equivalent to the linear program (LP) in $\phi(\mathbf{X};\omega)$:

$$\begin{aligned}
\omega^* = & \operatorname*{arginf}_{\omega \in \Omega} Ct - \langle \mathbf{r}, \phi(X;\omega) \rangle \\
& \text{s.t.} \\
& \quad At - \langle \boldsymbol{\eta}, \phi(X;\omega) \rangle = 1, \\
& \quad t > 0, \\
& \quad Ct > \langle \mathbf{r}, \phi(X;\omega) \rangle.
\end{aligned}$$

In Summary our problem is a quasi-convex function of a nonlinear function of ω. While using the ReLU activation, we have a quasi-convex function of a convex function of $\omega \equiv (\mathbf{u}, b)$. Next, we present a few ways to solve this problem.

Gradient Descent

Taking logs (9.16) becomes

$$\begin{aligned}
\omega^* &= \operatorname*{arginf}_{\omega \in \Omega} \log(C - c(\omega)) - \log(A - a(\omega)) \quad \text{s.t.} \quad a(\omega) \leqslant A \\
&= \operatorname*{arginf}_{\omega \in \Omega} \log(C - \langle \mathbf{r}, \phi(\mathbf{X};\omega) \rangle) - \log(A - \langle \boldsymbol{\eta}, \phi(\mathbf{X};\omega) \rangle) \\
& \quad \text{s.t.} \quad \langle \boldsymbol{\eta}, \phi(\mathbf{X};\omega) \rangle \leqslant A.
\end{aligned} \tag{9.20}$$

While the problem is non-convex, fortunately it is differentiable in ω. We could minimize just by gradient descent in ω.

An Approximation

If we substitute the first order approximation of $\log(1 + x) \approx x$ in 9.20, we get

$$\omega^* = \operatorname*{argsup}_{\omega} \Big\langle \phi(\mathbf{X};\omega), \mathbf{r} - \tfrac{C}{A}\boldsymbol{\eta} \Big\rangle \quad \text{s.t.} \quad \langle \boldsymbol{\eta}, \phi(\mathbf{X};\omega) \rangle \leqslant A \tag{9.21}$$

Without the $-\frac{C}{A}\boldsymbol{\eta}$ the above modified problem is equivalent to finding the next predictor with the highest covariance to the current residuals. Where as, $r - \frac{C}{A}\boldsymbol{\eta}$ is the residuals of the full-fit using the current active set. Thus this approximation is the forward step-wise regression with infinite predictors. In Figure 11.1 we present a hypothetical instance where we have an active set whose covariance with the active set starts at C and goes down with a slope A as we add γ amount of $\boldsymbol{\eta}$ to the fit. We present four variables and their negations, each of them have different initial covariances (c_j) with the active set and decay at different rates (a_j) as we add more $\boldsymbol{\eta}$ to the fit. While Variable 4 is preferred when $\gamma = 0$, when we add C/A of fit, we see that Variable 1 has the highest covariance with the residuals. Although we are looking for Variable 3 which is the one that will catch first when we add the correct amount of fit. This demonstrates the effect of this approximation. We can imagine a similar thing happening in infinite dimensions.

Quasi Gradient Descent

We note that problem (9.16), is a *Piecewise-Linear Fractional Function*, i.e. it would be a Linear Fractional Function except that it has a non-linearity. We know that LFFs are quasi-convex and we can used Quasi Gradient Descent. This problem is quasi-convex in $\phi(\mathbf{X};\omega)$. We use the chain-rule to get a quasi-gradient w.r.to ω and perform a quasi-gradient descent. The quasi-gradient w.r.to ϕ is

$$q(\phi) = \frac{\boldsymbol{\eta}}{A - \langle \boldsymbol{\eta}, \phi(X;\omega)\rangle} - \frac{\mathbf{r}}{C - \langle \mathbf{r}, \phi(X;\omega)\rangle} \tag{9.22}$$

Applying chain-rule, the pseudo-quasi-gradient w.r.to ω is

$$q(\omega) = \mathbf{J}(\omega)^T \left(\frac{\boldsymbol{\eta}}{A - \langle \boldsymbol{\eta}, \phi(X;\omega)\rangle} - \frac{\mathbf{r}}{C - \langle \mathbf{r}, \phi(X;\omega)\rangle} \right), \tag{9.23}$$

where $\mathbf{J}$ is the Jacobian in (5.8).

We see that this is FBly the gradient of the logged objective in (9.20). Hence we can think of the gradient descent algorithm to be a gradient descent on the logged objective (9.20) or a pseudo-quasi gradient descent on the original objective (9.7).

Iterative LP

As we have seen above the problem of finding the next ω to catchup can be formulated as a Linear Program in the $\phi(\mathbf{X};\omega)$.

$$\begin{aligned}
\omega^* = &\operatorname*{arginf}_{\omega\in\Omega} Ct - \langle \mathbf{r}, \phi(X;\omega)\rangle \\
&\text{s.t.} \\
&\quad At - \langle \boldsymbol{\eta}, \phi(X;\omega)\rangle = 1, \\
&\quad t > 0, \\
&\quad Ct > \langle \mathbf{r}, \phi(X;\omega)\rangle.
\end{aligned}$$

Unfortunately it is not an LP in ω. How ever by linearizing $\phi(\mathbf{X};\omega)$ around the current iterate as

$$\hat{\phi}(\mathbf{X};\omega) = \phi(\mathbf{X};\omega_0) + \mathbf{J}(\omega_0)^T(\omega - \omega_0) \tag{9.24}$$

our problem (9.16) becomes a Linear Fractional Function like (9.18) and can be written as an LP. This LP can be solved to get the next iterate ω_1. This can be repeated to get an estimate for ω^*.

Once we have the next ω that will catch up found using one of the two above methods, we can add it to the active set, step in the right direction of the fit (and the coefficients) to move to the next knot. Algorithm 8 describes this procedure in detail. It is identical to the finite dimensional version Algorithm 7.

Complications

Although in practice the LARs algorithm seems straight forward enough, in practice we encounter a lot of problems. Specifically the hard subproblem is solved via quasi gradient descent described above. In addition to the usual problem of random starting points, and stepsize selections, we have many more problems here. Firstly, the active set at a given iteration is in the domain of the subproblem (9.1.3). At these points of $\omega \in \mathcal{A}$, the objective we are trying to minimize is 0/0! The function there looks similar to the function $x1/x2$ in 2D at the origin. That is the limit of the function at these indeterminate points depends on the direction in which we are approaching them. This makes gradient descent extremely tricky, especially since one of the limits might be a very highly desirable value. And this happens at multiple locations in the domain. Also, the objective's denominator is negative on possibly disconnected regions on the domain! There are also a lot of local minima, unlike for the hard subproblem in the first order methods. Also if we could not find the best predictor in an iteration, and accidentally run into it at a later time, we do not know what to do with it. To this end, we use a few safeguards:

- We add to the objective a logarithmic barrier [Boyd and Vandenberghe, 2004] to push the gradient away from the active set.
- We run many versions of the subroutine with different starting points.
- If at any point we find that a predictor has more correlation with the residuals than the active set itself, it means that we have missed it in previous iterations. So we go back an iteration, and try the sub-routine again with this predictor as the initial value.

This last safeguard could get extremely complicated, and might lead to cases where one has to backtrack fifty iterations because a very good predictor was missed! This could make the algorithm in its ideal form practically worthless. We present a simulation study that demonstrates that this method could indeed work, but we do not recommend it in practice.

9.1.4 Simulation Study

We perform a simulation study like we did for the first order methods before. The problem instance is generated as previously done in section 7.1. It being LARs, we use MSE loss. We use the ReLU activation function, but this time with $\Omega_{\mathbf{X}}$ as our constraint set. This choice flows smoothly with the LARs implementation; although Ω_2 can also be used. We show results of just one run. We show detailed plots of the coefficient plots. Nearly 200 predictors are added into the active set. The LARs algorithm comes with an additional Lasso modification, where in, any coefficient that is crossing zero is set to zero and removed from the active set. Since our definition of $\Omega_{\mathbf{X}}$ involves negation closure, this results in positive coefficients only.

Figure 9.2 shows the coefficient paths obtained using Algorithm 8. We see that both the algorithms seem to do well and the plots for training and test error look similar to the ones we had before for the first order methods.

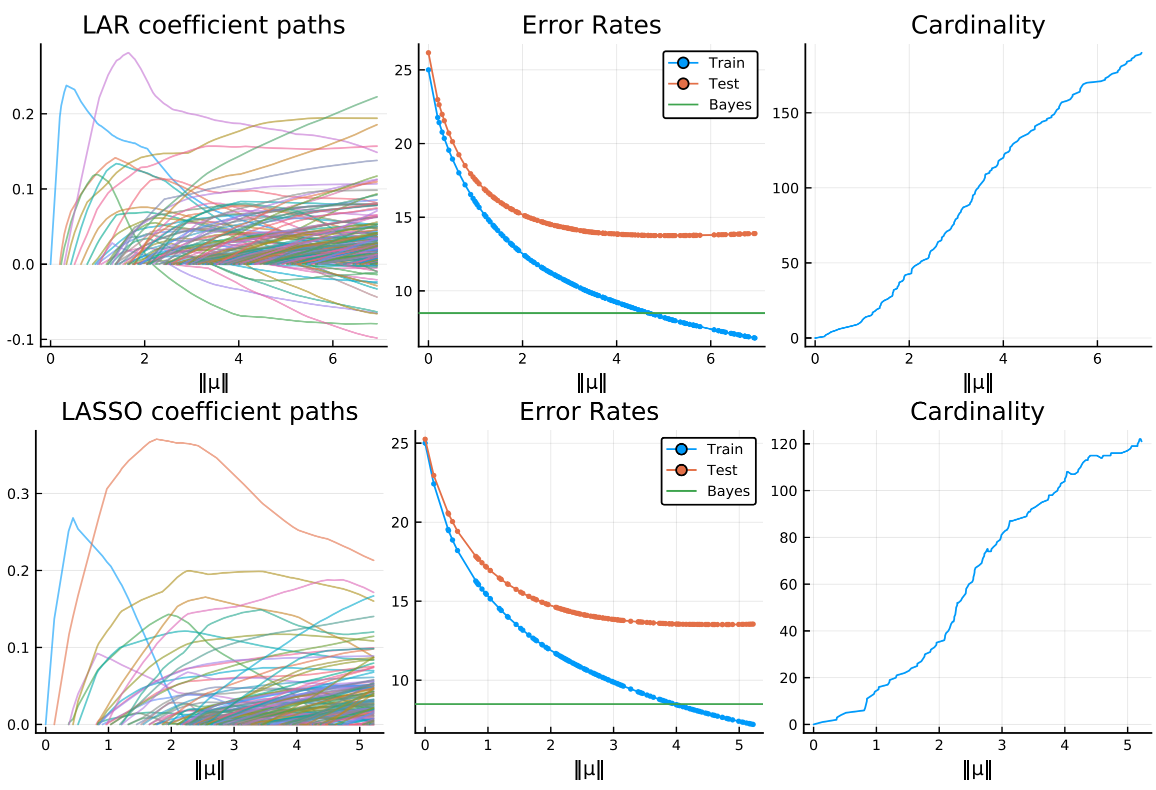

Figure 9.2: Coefficient paths for the infinite dimensional LARs. Top row is the LARs algorithm as is. The bottom row is the LARs algorithm with the LASSO modification.

9.2 Hessian Boosting

Here we will present a way to use the second derivative of the loss to find better predictors. Unlike the LARs algorithm presented above, this applies to any general loss, not just the MSE. This is a generalization of the very popular XGBoost algorithm in the framework of our problem (4.6). This means that we can get XGBoost algorithms for any choice of activation and constraint set pairs. We begin by a second order approximation to our objective.

For notational simplicity we define $f(\boldsymbol{\mu}) := \mathbf{L}(\Phi\boldsymbol{\mu}) := \mathbf{L}(\Phi\boldsymbol{\mu};\mathbf{y})$. Remember however that $\mathbf{L} : \mathbb{R}^n \mapsto \mathbb{R}$ where as $f : \mathbb{R}^\infty \mapsto \mathbb{R}$. The first order approximation to f at $\boldsymbol{\mu}_0$ is

$$\begin{aligned}\hat{f}_{\boldsymbol{\mu}_0} &= f(\boldsymbol{\mu}_0) + \nabla f(\boldsymbol{\mu}_0)^T(\boldsymbol{\mu} - \boldsymbol{\mu}_0) \\ &= \mathbf{L}(\Phi\boldsymbol{\mu}) + \nabla\mathbf{L}(\Phi\boldsymbol{\mu}_0)^T\Phi(\boldsymbol{\mu} - \boldsymbol{\mu}_0)\end{aligned}$$

and the second order approximation is

$$\begin{aligned}\hat{\hat{f}}_{\boldsymbol{\mu}_0} &= f(\boldsymbol{\mu}_0) + \nabla f(\boldsymbol{\mu}_0)^T(\boldsymbol{\mu} - \boldsymbol{\mu}_0) + \tfrac{1}{2}(\boldsymbol{\mu} - \boldsymbol{\mu}_0)^T\nabla^2 f(\boldsymbol{\mu}_0)(\boldsymbol{\mu} - \boldsymbol{\mu}_0) \\ &= \mathbf{L}(\Phi\boldsymbol{\mu}_0) + \nabla\mathbf{L}(\Phi\boldsymbol{\mu}_0)^T\Phi(\boldsymbol{\mu} - \boldsymbol{\mu}_0) + \tfrac{1}{2}(\Phi(\boldsymbol{\mu} - \boldsymbol{\mu}_0))^T\nabla^2\mathbf{L}(\Phi\boldsymbol{\mu}_0)\Phi(\boldsymbol{\mu} - \boldsymbol{\mu}_0) \\ &= \mathbf{g}_0^T\Phi\boldsymbol{\mu} - \mathbf{z}_0^T\mathbf{H}_0\Phi\boldsymbol{\mu} + \tfrac{1}{2}(\Phi\boldsymbol{\mu})^T\mathbf{H}_0\Phi\boldsymbol{\mu} + \text{const.},\end{aligned} \tag{9.25}$$

where $\mathbf{z}_0 = \Phi\boldsymbol{\mu}_0 \in \mathbb{R}^n$ is the fit, $\mathbf{g}_0 = \nabla_{\mathbf{z}}\mathbf{L}(\mathbf{z}_0;\mathbf{y}) \in \mathbb{R}^n$ is the gradient, and $\mathbf{H}_0 = \nabla^2_{\mathbf{z}}\mathbf{L}(\mathbf{z}_0;\mathbf{y}) \in \mathbb{R}^{n\times n}$ is the hessian of the the loss. Given that the losses we consider are decomposable into the loss from the n datapoints, the hessian is a diagonal matrix, with the i^{th} element given by the second derivative of the loss evaluated at the current fit for the i^{th} datapoint.

$$\mathbf{H}_{0ii} = \partial_z^2 L(z_i; y_i).$$

Minimizing the second order approximation is

$$\underset{\|\boldsymbol{\mu}\|_1 \leqslant C}{\operatorname{argmin}} \left\langle \Phi^T(\mathbf{g}_0 - \mathbf{H}_0\mathbf{z}_0), \boldsymbol{\mu}\right\rangle + \tfrac{1}{2}\left\langle \boldsymbol{\mu}, \Phi^T\mathbf{H}_0\Phi\boldsymbol{\mu}\right\rangle.$$

The normal equations for this problem (ignoring the norm restriction on $\boldsymbol{\mu}$ which will be enforced later via a projection) are

$$\begin{aligned}&\Phi^T(\mathbf{g}_0 - \mathbf{H}_0\mathbf{z}_0) + \Phi^T\mathbf{H}_0\Phi\boldsymbol{\mu} = 0 \\ \implies &\Phi^T\mathbf{g}_0 + \Phi^T\mathbf{H}_0\Phi(\boldsymbol{\mu} - \boldsymbol{\mu}_0) = 0 \\ \implies &\Phi^T\mathbf{g}_0 + \Phi^T\mathbf{H}_0\Phi\,\Delta\boldsymbol{\mu} = 0.\end{aligned}$$

This means that the optimal $\Delta\boldsymbol{\mu}$ solves the weighted non-linear least-squares problem:

$$\Delta\boldsymbol{\mu}^* = \underset{\Delta}{\operatorname{argmin}} \sum_{i=1}^{n} \mathbf{H}_{0ii} \left(\Phi\Delta + \frac{\mathbf{g}_{0i}}{\mathbf{H}_{0ii}} \right)^2 \tag{9.26}$$

$$= \underset{\Delta}{\operatorname{argmin}} \left\| \mathbf{H}_0^{1/2} \left(-\mathbf{H}_0^{-1}\mathbf{g}_0 - \Phi\Delta \right) \right\|_2^2 \tag{9.27}$$

This is a very hard problem almost as difficult as our original problem. To make this problem tractable, we impose the atomicity condition on $\Delta\boldsymbol{\mu}$ as $\Delta\boldsymbol{\mu} = \delta_{\omega^*}$ for some ω^*. That is, we want to add only one atom at each iteration, just like we did for first order methods. This means the next predictor that we are going to add to the fit, $\phi(\mathbf{X};\omega^*)$, best solves the above weighted non-linear least-squares problem amongst all $\omega \in \Omega$.

$$\omega^* = \underset{\omega\in\Omega}{\operatorname{argmin}} \min_{\rho} \sum_{i=1}^{n} \mathbf{H}_{0ii} \left(\rho\phi(\mathbf{x}_i;\omega) + \frac{\mathbf{g}_{0i}}{\mathbf{H}_{0ii}} \right)^2 \tag{9.28}$$

$$= \underset{\omega\in\Omega}{\operatorname{argmin}} \min_{\rho} \left\| \mathbf{H}_0^{1/2} \left(-\mathbf{H}_0^{-1}\mathbf{g}_0 - \rho\phi(\mathbf{X};\omega) \right) \right\|_2^2 \tag{9.29}$$

This is very similar to the Gradient Boosting subproblem with $\Omega_{\mathbf{X}}$ given by (5.35). Here we find the next best ω that is 'close' to the generalized residual (scaled by the second derivative), but now in a weighted sense — where the weights also come from the second derivative. Where some form of homogeneity of ϕ applies the above problem is solved up to a multiplicative factor on ω^*, which can then be scaled down to belong to Ω. Once we have the new active set, the corresponding weights can be calculated via Frank-Wolfe or via ϵ or line-search boosting.

We see that XGBoost is nothing but a specific implementation of this algorithm for the activation ϕ as trees and line-search for finding the size of the coefficient for the new member of the active set. Thus we have expanded the scope of XGBoost to any convex neural network, and in fact, to any sparse inverse problem where the above weighted least squares problem can be solved efficiently. In the case of trees, a greedily fit tree is considered a good enough approximation to the global solution of (9.28).

In addition, we could also use alternating descent to get leaner ensembles in conjunction with any of our algorithms where the subproblem is solved in a weighted manner as in (9.28). For e.g., we could have the machine learning algorithm AD-XGBoost with ReLUs. This idea looks very promising, and in the interest of time, we delay exploring it to future research. All the various combinations of algorithms, activations and constraint-sets can be applied/tried here. That makes it a large body of work in itself. While Ω_1 from section 6.4 was not a particularly good choice for first order methods, we feel that in this case, in conjunction with the weights, it could lead to directions of projection that are indeed sparse.

9.3 Infinite Dimensional Co-ordinate Descent

In this section we propose (without discussion or further study) yet another method that can be used to solve our main problem (4.6). Like Hessian Boosting from the above section this algorithm also works for any loss. Co-ordinate Descent (CoD) is the goto method to solve finite dimensional lasso problems via the very popular R package `glmnet` [Friedman et al., 2010]. Further theoretical properties can be found in [Friedman et al., 2007] and [Tseng, 2001]. CoD solves the problem in its lagrangian form given by

$$\min_{\boldsymbol{\mu}} \mathbf{L}\left(\Phi\boldsymbol{\mu};\mathbf{y}\right) + \lambda \left\|\boldsymbol{\mu}\right\|_1 \ , \tag{9.30}$$

where

$$\Phi\boldsymbol{\mu} = \int_\Omega \phi(\mathbf{X};\omega)d\boldsymbol{\mu}(\omega)$$

and

$$\left\|\boldsymbol{\mu}\right\|_1 = \left|\boldsymbol{\mu}\right|(\Omega) = \int_\Omega d|\boldsymbol{\mu}|(\omega).$$

Quadratic Approximation

Here we start with the quadratic approximation of the loss from Hessian Boost given by (9.25). Again we notate $f(\boldsymbol{\mu}) := \mathbf{L}(\Phi\boldsymbol{\mu}) := \mathbf{L}\left(\Phi\boldsymbol{\mu};\mathbf{y}\right)$. The approximation at $\boldsymbol{\mu}_0$ is given by:

$$\hat{\hat{f}}_{\boldsymbol{\mu}_0} = \mathbf{g}_0^T\Phi\boldsymbol{\mu} - \mathbf{z}_0^T\mathbf{H}_0\Phi\boldsymbol{\mu} + \tfrac{1}{2}(\Phi\boldsymbol{\mu})^T\mathbf{H}_0\Phi\boldsymbol{\mu} + \text{const.}, \tag{9.31}$$

where $\mathbf{z}_0 = \Phi\boldsymbol{\mu}_0 \in \mathbb{R}^n$ is the fit, $\mathbf{g}_0 = \nabla_{\mathbf{z}}\mathbf{L}\left(\mathbf{z}_0;\mathbf{y}\right) \in \mathbb{R}^n$ is the gradient, and $\mathbf{H}_0 = \nabla^2_{\mathbf{z}}\mathbf{L}\left(\mathbf{z}_0;\mathbf{y}\right) \in \mathbb{R}^{n\times n}$ is the hessian of the the loss. The hessian is a diagonal matrix defined by $\mathbf{H}_{0ii} = \partial_z^2 L\left(z_i;y_i\right)$.

When minimizing this second order approximation with our penalty in the lagrangian form, our problem (9.30) becomes

$$\operatorname*{argmin}_{\boldsymbol{\mu}} \left\langle \Phi^T(\mathbf{g}_0 - \mathbf{H}_0\mathbf{z}_0), \boldsymbol{\mu}\right\rangle + \tfrac{1}{2}\left\langle \boldsymbol{\mu}, \Phi^T\mathbf{H}_0\Phi\boldsymbol{\mu}\right\rangle + \lambda\left\|\boldsymbol{\mu}\right\|_1. \tag{9.32}$$

This is the weighted lasso subproblem from the Iterative Reweighted Least Squares procedure. The working response is $\mathbf{z}_0 - \mathbf{H}_0^{-1}\mathbf{g}_0$ and the weights are $\operatorname{diag}\left(\mathbf{H}_0\right)$. When we are in the finite dimensional case (with $\Omega = [p]$ and $\mathbf{X} = \Phi$), we recover the approximation from [Friedman et al., 2010], where coordinate descent is used to solve this quadratic approximation.

However, in our general case we can not use the coordinate descent as is, for the obvious reason that we can not cycle through a dense set of coordinate indices given by $\omega \in \Omega$ like we did when $j \in \{1, 2, \cdots, p\}$. Now we present a way to do coordinate descent in infinite dimensions.

Coordinate Descent

CoD is used to obtain solutions over a path of λ values, starting from the highest to the lowest. The highest value of λ is calculated like in the case of LARs.

$$\lambda_0 = \max_j \langle \mathbf{y}, \mathbf{X}_j \rangle,$$

which in the infinite dimensional case becomes

$$\lambda_0 = \max_{\omega \in \Omega} \langle \mathbf{y}, \phi(\mathbf{X}, \omega) \rangle.$$

This initial step is just an instance of the hard subproblem and is easily solved via the methods described in Chapter 6. The rest of the λ's are usually chosen over a logarithmic grid going down. Warm starts are used to quickly obtain solutions at the next value of λ. This pathwise method is quicker than computing solutions at a given λ directly. The CoD method for a general loss $\mathbf{L}$ has the three main loops:

1. Outer Loop: Decrement value of λ
2. Middle Loop: Obtain a quadratic approximation of loss at current co-efficient as given by (9.32)
3. Inner Loop: Descend on each co-ordinate to solve the penalized weighted least squares problem (9.32)

As mentioned above the inner most loop is intractable in the infinite dimensional case, here we will present an alternate.

In the finite dimensional case, the update for a coefficient β_j is

$$\beta_j \leftarrow \frac{S_\lambda\left(\sum_{i=1}^n w_i x_{ij}(r_i - r_i^{(j)})\right)}{\sum_{i=1}^n w_i x_{ij}^2} \tag{9.33}$$

Here S_λ is the soft-thresholding proximal operator for the ℓ_1 norm given by

$$S_\lambda(z) = \begin{cases} z - \lambda & \text{if} \quad z > \lambda \\ z + \lambda & \text{if} \quad z < -\lambda \\ 0 & \text{otherwise.} \end{cases} \tag{9.34}$$

$\mathbf{x}_i$ are the datapoints. $\mathbf{r} = \mathbf{z}_0 - \mathbf{H}_0^{-1}\mathbf{g}_0$ is the working response. $\mathbf{r}^{(j)} = \sum_{l \neq j} \mathbf{X}_l \beta_l$ is the fitted value excluding the contribution from $\mathbf{X}_j$. Hence $\mathbf{r} - \mathbf{r}^{(j)}$ is the partial residuals.

In the inner loop, we cycle through the active set and apply the above soft-thresholding operation

(9.33) till convergence. Then we check the KKT conditions on the inactive predictors.

$$\mathcal{A}(\lambda) = \{j \mid \beta_j(\lambda) \neq 0\} \subset \{j \mid |\langle X_j, \mathbf{g}(\lambda)\rangle| = \lambda\}.$$

If none of the inactive predictors satisfy the condition for a non-zero coefficient, we quit the inner loop. Else, the violating contenders are added to the active set and the inner-most loop is repeated.

Coordinate Descent in Infinite Dimensions

The inner loop remains the same for the active set. Remember that the active set begins with

$$\lambda = \lambda_0 = \max_{\omega\in\Omega} \langle \mathbf{y}, \phi(\mathbf{X}, \omega)\rangle$$
$$\mathcal{A}(\lambda_0) = \left\{ \underset{\omega\in\Omega}{\operatorname{argmax}} \langle \mathbf{y}, \phi(\mathbf{X}, \omega)\rangle \right\}.$$

At any value of the atomic coefficient measure,

$$\boldsymbol{\mu}(\cdot) = \sum_{j=1}^{m} \beta_j \delta_{\omega_j}(\cdot), \tag{9.35}$$

we apply on the active set the soft-thresholding operation (9.33) in a cyclical manner till convergence (like we did in the finite dimensional case). Now while checking for violators of the KKT conditions

$$\mathcal{A}(\lambda) = \{\omega \in \Omega \mid \boldsymbol{\mu}(\{\omega\}; \lambda) \neq 0\} \subset \{\omega \in \Omega \mid |\langle \phi(\mathbf{X}; \omega), \mathbf{g}(\lambda)\rangle| = \lambda\}$$

on the inactive predictors, we note that we can not cycle through the usually dense set Ω. So we now solve the hard subproblem of chapter 6 (given by (6.1))

$$\omega^* = \underset{\omega\in\Omega}{\operatorname{argmin}} \langle \mathbf{g}, \phi(\mathbf{X}; \omega)\rangle, \tag{9.36}$$

or more familiarly when negation closure does not apply,

$$\omega^* = \underset{\omega\in\Omega}{\operatorname{argmax}} |\langle \mathbf{g}, \phi(\mathbf{X}; \omega)\rangle|. \tag{9.37}$$

When the new predictor $\phi(\mathbf{X}; \omega^*)$ satisfies the KKT condition for a non-zero coefficient we add it to the active set and repeat the inner loop till convergence. This process is repeated till the next best ω is not a contender for a non-zero coefficient, i.e.

$$\max_{\omega\in\Omega} |\langle \mathbf{g}, \phi(\mathbf{X}; \omega)\rangle| < \lambda. \tag{9.38}$$

Remarks

We have just presented a way to use the highly popular coordinate descent algorithm in infinite dimensions. We have not tested this problem, but feel that it is a serious contender and is a topic of future research. This algorithm results in the infinite dimensional version of `glmnet`. At each value of λ along the path, we get an ensemble like we do for all the other methods. However, we might keep getting new ω^*'s from (9.37), or even worse, we might keep getting ones that are very close to the active set, leading us to have to drop the old ω's for their neighbours. Caratheodory's only guarantees that we need no more than $n+1$, which is still a lot. To circumvent this problem, we could use an early stopping criterion and quit when the training error does not go down significantly with a new predictor. This approximation only adds to the approximation in solving the hard subproblem in infinite dimensions.

The hard subproblem remains the same for a given choice of activation-constraint pair (ϕ, Ω). Hence this method can be easily extended by modifying the code from a good implementation of `glmnet` as described above. The outer loops remain the same only the KKT check on the inner loop changes. This check uses the hard subproblem solvers of chapter 6. This extends the scope of `glmnet` to solve any sparse inverse problem reasonably. Where the hard subproblem is solvable satisfactorily.

We also feel that alternating descent helps greatly in this context, reducing the number of new predictors, especially neighbours of existing ones coming in and delaying the fitting. Again, this algorithm can be implemented and analyzed for the various combinations of losses, activations and constraint-sets and is a body of work in itself.

Chapter 10

Further Regularization

In this section we will explore a few ideas to give further regularization in our model.

10.1 Saturating ReLUs

Unlike trees that take on only a fixed set of outputs, a ReLU (and its relatives) can take on any unbounded value. This could be a serious issue when it comes to outliers especially in the test data. Hence it might make sense to saturate the ReLU, so that it takes on values not more than what have been observed on the training data. This idea is explored more formally but with generalized addivite models in [Boyd et al., 2016].

10.2 Elastic-Net Style Convolved Measures

We know that, in the case of traditional lasso, when there is very high correlation between the predictors, the lasso picks one of them arbitrarily. In our case this problem is taken to the extreme, since the correlation between two predictors indexed by ω's that are very close to each other can get arbitrarily close to perfection. Elastic net is known to work well in such cases with high correlation. It adds a bunch of highly correlated predictors together where a lasso would add only anyone of them. We could simulate the same behaviour for our problem.

Instead of augmenting the coefficient measure $\boldsymbol{\mu}$ by an atom at a particular ω, we could add to it a kernel around that ω with a predetermined width. This naturally extends the scope of our fits from just atomic measures, to measures that are convolutions of atomic measures with a predetermined kernel. We will now see a concrete example.

10.2.1 Example: ReLU with Gaussian Kernel

Consider the case of the ReLU activation and a gaussian kernel of fixed width σ. We have the ReLU:

$$\phi(\mathbf{x}, \mathbf{u}, b) := \max(0, \mathbf{x}^T\mathbf{u} + b) := (\mathbf{x}^T\mathbf{u} + b)_+. \tag{10.1}$$

The convolution of this activation function (a single atom measure) with the gaussian kernel will give us the following function, which is equivalent to evaluating the ReLU in a neighbourhood around $\mathbf{u}$ and weighing it with a Gaussian Kernel.

$$\begin{aligned}\hat{\phi}_\sigma(\mathbf{x}, \mathbf{u}, b) &= \int_{\mathbf{w}\in\mathbb{R}^p} \phi(\mathbf{x}, \mathbf{w}, b) d\mathcal{N}\left(\frac{\mathbf{w}-\mathbf{u}}{\sigma}\right) \\ &= \frac{1}{(\sqrt{2\pi}\sigma)^p} \int_{\mathbf{w}\in\mathbb{R}^p} \phi(\mathbf{x}, \mathbf{w}, b) \exp\left(-\tfrac{1}{2\sigma^2}\left\|\mathbf{w}-\mathbf{u}\right\|_2^2\right) d\mathbf{w} \\ &= \frac{1}{(\sqrt{2\pi}\sigma)^p} \int_{\mathbf{w}\in\mathbb{R}^p} (\mathbf{x}^T\mathbf{w} + b)_+ \exp\left(-\tfrac{1}{2\sigma^2}\left\|\mathbf{w}-\mathbf{u}\right\|_2^2\right) d\mathbf{w}\end{aligned}$$

Define a matrix $\mathbf{J}$ (for Jacobian) such that the first column of $\mathbf{J}$ is the unit norm vector $\mathbf{x}/\left\|\mathbf{x}\right\|_2$, and the rest are ortho-normal to $\mathbf{x}$. i.e.

$$\mathbf{J}^T\mathbf{x} = \left\|\mathbf{x}\right\|_2 e_1,$$

where e_1 the first cardinal direction. Substituting for $\mathbf{w} = \mathbf{J}\mathbf{v}$ in the above,

$$\begin{aligned}\hat{\phi}_\sigma(\mathbf{x}, \mathbf{u}, b) &= \frac{1}{(\sqrt{2\pi}\sigma)^p} \int_{\mathbf{v}\in\mathbb{R}^p} (\mathbf{x}^T\mathbf{J}\mathbf{v} + b)_+ \exp\left(-\tfrac{1}{2\sigma^2}\left\|\mathbf{J}\mathbf{v}-\mathbf{J}\mathbf{J}^T\mathbf{u}\right\|_2^2\right) |\mathbf{J}|\, d\mathbf{v} \\ &= \frac{1}{(\sqrt{2\pi}\sigma)^p} \int_{\mathbf{v}\in\mathbb{R}^p} (\left\|\mathbf{x}\right\|_2 e_1^T\mathbf{v} + b)_+ \exp\left(-\tfrac{1}{2\sigma^2}\left\|\mathbf{v}-\mathbf{J}^T\mathbf{u}\right\|_2^2\right) d\mathbf{v} \\ &= \frac{1}{(\sqrt{2\pi}\sigma)^p} \int_{\mathbf{v}\in\mathbb{R}^p} (\left\|\mathbf{x}\right\|_2 v_1 + b)_+ \exp\left(-\tfrac{1}{2\sigma^2}\left\|\mathbf{v}-\mathbf{J}^T\mathbf{u}\right\|_2^2\right) d\mathbf{v}\end{aligned}$$

Here we used the facts that $\mathbf{J}\mathbf{J}^T = \mathbf{I}$; $|\mathbf{J}| = 1$; and that rotation does not change norm. The integral separates now into the individual components of $\mathbf{v} = (v_1, v_2, \cdots, v_p)$. All except v_1 integrate out to 1, leaving us with

$$\hat{\phi}_\sigma(\mathbf{x}, \mathbf{u}, b) = \frac{1}{\sqrt{2\pi}\sigma} \int_{v_1\in\mathbb{R}} (\left\|\mathbf{x}\right\|_2 v_1 + b)_+ \exp\left(-\tfrac{1}{2\sigma^2}(v_1 - \mathbf{x}^T\mathbf{u}/\left\|\mathbf{x}\right\|_2)^2\right) dv_1 \tag{10.2}$$

Now, taking $v = \|\mathbf{x}\|_2 v_1 + b$,

$$\hat{\phi}_\sigma(\mathbf{x}, \mathbf{u}, b) = \frac{1}{\sqrt{2\pi}\sigma} \int_{v \in \mathbb{R}} (v)_+ \exp\left(-\tfrac{1}{2\sigma^2 \|\mathbf{x}\|_2^2} (v - b - \mathbf{x}^T\mathbf{u})^2\right) \frac{dv}{\|\mathbf{x}\|_2} \tag{10.3}$$

$$= \frac{1}{\sqrt{2\pi}\sigma \|\mathbf{x}\|_2} \int_{v \in \mathbb{R}} v \exp\left(-\tfrac{1}{2\sigma^2 \|\mathbf{x}\|_2^2} (v - b - \mathbf{x}^T\mathbf{u})^2\right) \mathbb{I}(v > 0)\, dv \tag{10.4}$$

This is the Expectation of the *Truncated Gaussian Random Variable* $\mathcal{N}(\mathbf{x}^T\mathbf{u} + b, \sigma^2 \|\mathbf{x}\|_2^2)$ truncated at 0. By Mill's Ratio, we have this to be:

$$\hat{\phi}_\sigma(\mathbf{x}, \mathbf{u}, b) = \mathbb{E}(V \mid V > 0) \quad \text{where} \quad V \sim \mathcal{N}\left(\mathbf{x}^T\mathbf{u} + b, \sigma^2 \|\mathbf{x}\|_2^2\right) \tag{10.5}$$

$$= (\mathbf{x}^T\mathbf{u} + b) + \sigma \|\mathbf{x}\|_2 \frac{\varphi(\frac{-b - \mathbf{x}^T\mathbf{u}}{\sigma\|\mathbf{x}\|_2})}{1 - \Phi(\frac{-b - \mathbf{x}^T\mathbf{u}}{\sigma\|\mathbf{x}\|_2})} \tag{10.6}$$

$$= (\mathbf{x}^T\mathbf{u} + b) + \sigma \|\mathbf{x}\|_2 \frac{\varphi(\frac{\mathbf{x}^T\mathbf{u} + b}{\sigma\|\mathbf{x}\|_2})}{\Phi(\frac{\mathbf{x}^T\mathbf{u} + b}{\sigma\|\mathbf{x}\|_2})} \tag{10.7}$$

where φ, Φ are the PDF and CDF of the standard normal respectively.

Given this closed form expression, we can regularize an ensemble after it is completely fit, by replacing each atom with a gaussian kernel of fixed width around it. Or this smoothing can be incorporated during training, by adding to the ensemble the kernel (instead of the atom) before seeking the next best predictor.

Chapter 11

Discussion

In this chapter we look at the various algorithms and the subroutines together. As previously mentioned, this body of work lets us explore the connections between popular methods like Gradient Boosting and well-understood algorithms like Frank-Wolfe. We present our main problem (4.6) as an ℓ_1 penalized convex neural network with a motivation to get a good ensemble fit that is both sparse and that generalizes well. The ℓ_1 penalty allows us to keep the problem convex, letting us apply Frank-Wolfe — a method with known convergence properties [Jaggi, 2013] (when the hard subproblem can be solved well). At the same time the ℓ_1 penalty encourages sparsity. Our analysis of the problem and the observation of the various methods has given some some new perspectives on boosting and its relatives.

11.1 New Perspectives

Now we will present new perspectives on boosting, especially with regards to how it relates to methods like Frank-Wolfe and LARs. Focus will be on the amount of $\|\boldsymbol{\mu}\|_1$ that is allocated to old predictors vs the new predictor.

Holistic Perspective on Gradient Boosting

In the process of trying to solve our problem, we discover that Gradient Boosting is a popular way to approximately solve the same problem. This allows us to look at GB in a new light as previously touched upon in [Rosset et al., 2004, Friedman, 2012]. This is in contrast to the more conventional view that Boosting is trying to find predictors that are close to the residual in a squared-error sense. We now clearly see that what are the residuals are indeed the negative of the gradient of the loss with respect to the fit. Secondly, we also see that the closeness in the squared-error sense is only in the case of $\Omega_{\mathbf{X}}$ (as implicitly assumed by both [Friedman, 2012, Rosset et al., 2004]). But in general we are trying to maximize the absolute inner product between the negative gradient and the next predictor of interest. Further we see that this act of trying to maximize the absolute inner product, is nothing but minimizing a linearization of our loss, which indeed leads us to a vertex, giving us the next predictor to be added to the active set.

Amount of New Predictor to be Added

How much of the new predictor we add to the fit differs from algorithm to algorithm. The amount of addition is again informed by the ℓ_1 penalty. We also realize that greedily adding as much of the new predictor as we can is not the best approach — as evidenced by the fact that damping is so widely used in practical application of gradient boosting. A much better way of approaching it seems to be ϵ-Boosting or Stagewise as we call it. Where ϵ amount of the best predictor is being added to the fit at each time. Although this takes longer to fit, we see from the studies that it both generalizes well and is leaner. While FW takes this step much more seriously and does a full finite dimensional lasso on the active set. We find it less appealing overall, as it is time consuming, and does not result in a much better fit. Also FW is a point algorithm and the whole process needs to be repeated each time we increase the bound on $\|\boldsymbol{\mu}\|_1$.

Amount of Old Predictors before New one Comes In

Now we will compare the path algorithms on the basis of how far they go along the path before they let the next predictor in. At each iteration, Gradient Boosting only adds the new predictor to the fit; and before the next predictor can come in, it adds as much of the last one to come in as it can. Stagewise imitates LARs here and keeps adding the old ones just enough until the new one is in a comparable light. In this regard, it tries to emulate LARs. LARs tries to be precise about this and makes the hard subproblem harder by trying find both (γ^*, ω^*) — where γ^* is the amount of the old active set to be added to the fit before ω^* enters the fit. Given the closeness between LARs and SW, an interesting modification to SW could be adding ϵ amount of the equi-angular direction $\boldsymbol{\eta}$ to the fit rather than ϵ amount of just one of the active set. This could greatly reduce the computation time of Stagewise and is a good direction for future research.

In this regard, in Figure 11.1, we present a sample scenario for MSE loss and see which predictors are preferred by which methods. There is the current active set, all members of which have a correlation of 5 with the residuals. There are four other predictors notated ϕ_j with a correlation of j with the residuals. As we add γ amount of the vector $\boldsymbol{\eta}$ that is equiangular to the active set, the correlation of the active set with the residuals keeps coming down linearly (at a rate of unity). For the other predictors, the rate of decay is different. For e.g., ϕ_3 is orthogonal to the active set and hence its correlation to the fit does not change as $\boldsymbol{\eta}$ is added to the fit. Similarly, the other predictors behave in various fashions. In the LARs scenario, where we are trying to solve our problem exactly, ϕ_3 is the desired predictor. However, if we were using gradient boosting with a lot of damping, we

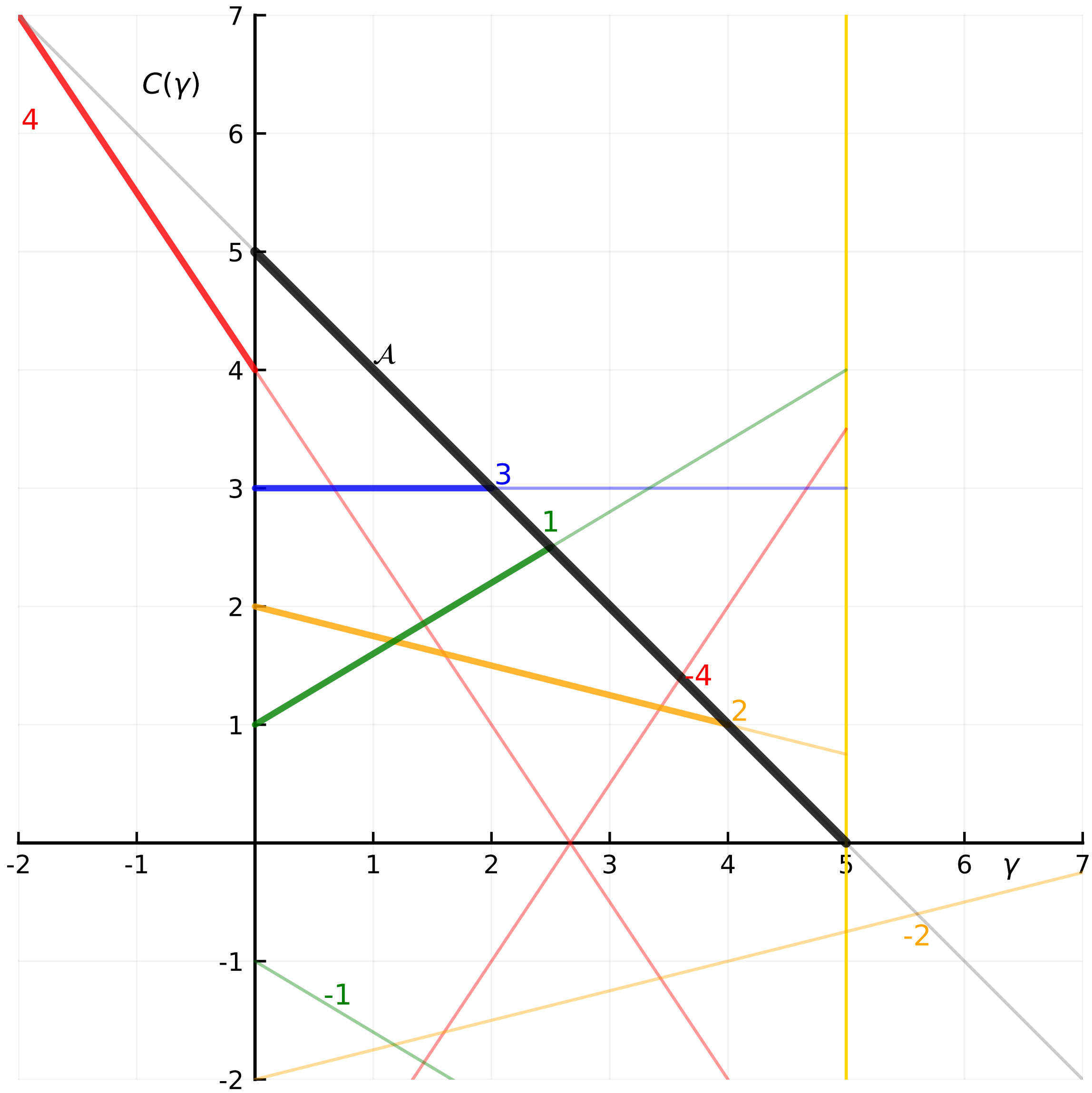

Figure 11.1: Covariances (with residuals $\mathbf{r}$) of four predictors $\{\phi_1, \phi_2, \phi_3, \phi_4\}$ (and their negations) in comparison to that of the active set $\phi_{\mathcal{A}}$. As we add γ amount of the equiangular direction $\boldsymbol{\eta}$ to the fit, the covariance decreases at different rates for each of the predictors. If we were using Gradient Boosting with heavy damping, ϕ_4 would be of interest as it has the highest covariance with $\mathbf{r}$. ϕ_3 catches up first and hence is of interest in LARs. ϕ_1 is picked in the forward step-wise scenario, or in gradient boosting without damping. The behavior of ϕ_2 is what we see most commonly in practice.

look for the next predictor which best correlates to the predictors, which in this case will be ϕ_4 — not a good choice. If we are choosing predictors in a forward step-wise fashion (possibly in an attempt to be really sparse), we add all of five units $\boldsymbol{\eta}$ till the active set has no correlation with the residuals, at this point ϕ_1 has the best correlation to the residuals and is preferred. This is akin to the scenario of boosting with no damping. Forward stagewise in an attempt to mimic the behaviour of LARs, while staying applicable to general losses, keeps adding the active set to the fit until ϕ_3 becomes a contender. Overall, ϕ_4 is a bad contender and we look for predictors like ϕ_3 which become good predictors down the path. This explains why damping is such an important parameter in Gradient Boosting: as we do not know when the next predictor will catchup, we will have to damp by the correct amount to capture ϕ_3 into the active set next.

11.2 Contributions

Our work falls at the intersection of three fields of research: Neural Networks, Sparse Inverse Problems and Boosting. More specifically, we formulate our problem as an infinite dimensional Neural Network with an ℓ_1 penalty on the weights. The same ℓ_1 penalty makes our problem an instance of a sparse inverse problem. Finally it also generalizes the boosting framework of [Friedman, 2012] to uncountably many predictors. We summarize our contributions from each of these perspectives. See Figure 11.2 for a summary.

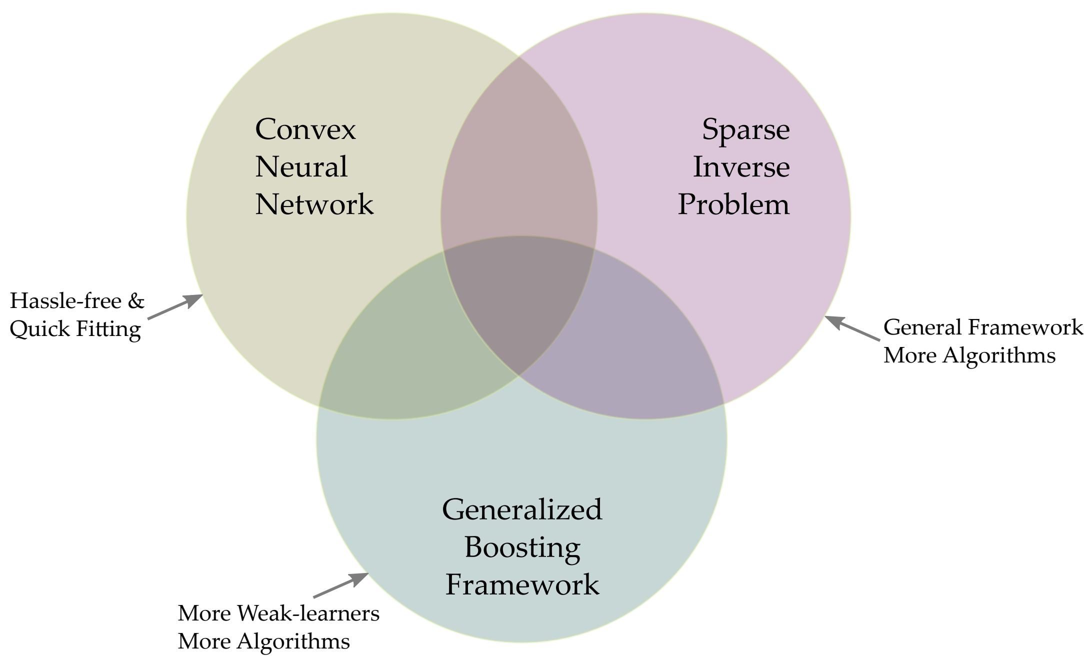

Figure 11.2: Our work at the intersection of three fields, and our contributions form each perspective.

11.2.1 Neural Networks

Neural Networks are notoriously hard to train. Even with one hidden layer the number of hyper-parameters to be tuned include: the norm of $\boldsymbol{\mu}$, the stepsize for each of the layers, the number of hidden layer nodes, random initialization parameters, stepsize annealing, number of iterations, etc. Many times, a number of stepsize configurations have to be tried before the network starts to converge. By posing it as a convex neural network and using FW like optimization techniques, we build the network incrementally. This eliminates almost all the hyper-parameters. A comparison of hyper-parameters required is presented in Table 5.2. To extend this idea of easy fitting to deeper dense networks, once the first layer is fit, we can replace the input with the output of the first layer and repeat the process. This is akin to the deep forest idea of [Zhou and Feng, 2017], where successive layers of random forests are built on top of the outputs of previous layers.

11.2.2 Generalizing the Boosting Framework

We generalize the boosting framework of [Friedman, 2012] to dense Ω's and present alternate algorithms to the traditional gradient boosting. This increases the scope of Gradient Boosting in the following ways:

1. Boosting can now be done with any activation-constraint pair. Although boosting any activation is known. We generalize it to any constraint set Ω. So far $\Omega_{\mathbf{X}}$ seems to be the only viable option given that the next best predictor was obtained by solving the non-linear least-squares problem (5.35). We present a general way to solve the subproblem in chapter 6

2. We present alternative algorithms to traditional greedy gradient boosting. We see that other methods like Frank-Wolfe and SW-FB can have better performance with less hyper-parameter tuning than the conventional way of boosting. This also makes the fitting algorithm more amenable to theoretical analyses.

3. Now the idea of Alternating Descent can be applied to the GB algorithm to get leaner and better fits.

4. Now we can 'boost' any Sparse Inverse Problem as long as you can solve the (usually hard) subproblem (6.1), using any of the seven first order algorithms and four second order methods.

5. We also present you a couple of second order methods (and their AD counterparts): Hessian-Boost and ∞D coordinate descent that can be used to do weighted boosting, that is possibly better.

6. Our choice of ReLU with $\Omega 2$ allows us to pull the complexity of the weak-learner into that of the ensemble. This reduces the number of tuning parameters compared to XGBoost with

Trees by two. The complexity of trees needs the specification of a penalty on the number of terminal nodes and another on the weights.

Best Boosting Algorithm

Some where in the many choices of parameters of our problem lurks the best configuration for boosting. Gradient Boosting with trees gives us a piece-wise constant fit. But if you look at the partial dependence plots given by the large ensemble (with large number of regions in each individual tree), we see that the resulting fit is not stay constant for any reasonably sized window in the X space. Also, we know that Second Order Fused Lasso (also called Trend Filtering) is an asymptotically optimal method to give piecewise linear fit in 1D. These two facts motivate us to try Boosting configurations that result in piecewise linear fits in higher dimensions. There have been efforts in this direction using generalized additive models including, [Boyd et al., 2016, Sadhanala and Tibshirani, 2017]. There is scope to go beyond GAMs with our model. It remains to pick a good Ω with the ReLU activation. We have seen in section 6.4 that sparsity inducing Ω_1 does not work well with first order methods. As a remedy we propose the use of Ω_1 with the second order method HessianBoost to get a good weak-learners with sparse supports, thus making the ensemble robust to noise variables. We leave this as future work.

11.2.3 Sparse Inverse Problems

In chapter 8, we have presented a multiple-dispatch based Julia framework to solve our problem. This framework can be used to solve any Sparse Inverse Problem, many examples of which are given in [Boyd et al., 2015]. In addition to the ADFW algorithm presented in that reference, we can use all the seven first order methods and the four second order methods (HessianBoost, ∞D-CD and their AD counterparts) to solve your Sparse Inverse Problem of choice. We clearly outline in chapter 8 which sub-routines need to be coded for which of the algorithms to work with your Types of loss, activation and constraint set. Here we give a brief example of prior estimation from [Kiefer and Wolfowitz, 1956, Laird, 1978].

Prior Estimation for Empirical Bayes

Let each row of the observed data matrix $\mathbf{X}$ be generated as $X|\boldsymbol{\theta}_i \sim \mathcal{N}(\boldsymbol{\theta}_i, I_p)$. Where the mean parameters come from a prior $\boldsymbol{\theta} \sim F$. Given the n datapoints, we want to estimate the prior F. The unconditional likelihood of X is given by

$$f_F(\mathbf{x}) = \int_{\boldsymbol{\theta}} \phi(\mathbf{x} - \boldsymbol{\theta}) dF(\boldsymbol{\theta}). \tag{11.1}$$

The maximum likelihood problem of finding a good F, given the observations $\mathbf{X}$, becomes

$$\hat{F} = \underset{F}{\operatorname{argmax}} \sum_{i=1}^{n} \log(f_F(\mathbf{x}_i)) \quad \text{s.t.} \quad F \succeq 0, \quad \int_{\boldsymbol{\theta}} dF(\boldsymbol{\theta}) = 1. \tag{11.2}$$

This is an instance of a sparse inverse problem like our negation closed problem (4.17) with the positivity constraint on the measure. Massaging it into that form,

$$\hat{F} = \underset{F}{\operatorname{argmin}} \, \mathbf{L}(\Phi F) \quad \text{s.t.} \quad F \succeq 0, \;\; \|F\|_1 = 1. \tag{11.3}$$

Here the data remains the matrix $\mathbf{X}$. $\mathbf{y}$ is absent and is considered NULL. The measure $\boldsymbol{\mu}$ is now the prior F over $\mathbb{R}^p$. The 'activation' function ϕ is the standard gaussian pdf in p dimensions, parameterized by the mean $\boldsymbol{\theta}$, which is our ω. Ω is $\mathbb{R}^p$. The fit is the likelihood, and the loss is its negative log. When F is atomic, the fit $\mathbf{z}$ and the loss are given by

$$\begin{aligned}
F &\overset{\text{atom.}}{=} \sum_{j=1}^{m} \beta_j \delta_{\boldsymbol{\theta}_j} \quad \text{where} \quad \beta_j > 0 \\
z_i &= \sum_{j=1}^{m} \beta_j \, \phi(\mathbf{x}_i - \boldsymbol{\theta}_j) \\
L(z_i) &= -\log(z_i) \\
\partial_z L(z_i) &= -1/z_i
\end{aligned}$$

The hard subprob for this configuration of loss, activation and constraint-set becomes

$$\begin{aligned}
\omega^* &= \underset{\omega}{\operatorname{argmin}} \langle \nabla_{\mathbf{z}} \mathbf{L}, \phi(\mathbf{X}; \omega) \rangle \\
\implies \boldsymbol{\theta}^* &= \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{i=1}^{n} \frac{1}{z_i} \exp\left(-\tfrac{1}{2} \left\|\mathbf{x}_i - \boldsymbol{\theta}\right\|_2^2\right).
\end{aligned}$$

That is, in the end, our estimate of the prior is atomic, where the active set is built by solving the above subproblem, and the coefficients β_j are suggested by FW or SW. The subproblem is trying to find the mean parameter to lie close to the individual datapoints (in an exponentiated squared distance sense) weighted by the residual (inverse of the current likelihood estimate z_i). This problem can be solved using any of our methods.

11.3 Future Work

We conclude by mentioning a few directions for future work. As we have already mentioned many promising algorithms and choices of constraint-sets have to be tried. These include the sparsity inducing constraint-set Ω_1 of (4.12). Then there are the second order methods of chapter 9. A more tractable version of **LARs** might be possible. Instead of trying to solve the very hard LARs subproblem given by (9.1.3), we could take a jump along the current equi-angular direction $\boldsymbol{\eta}$ (9.4) by an amount of a fraction of C/A, defined in (9.10). This reduces the subproblem of LARs to the usual subproblem (6.1). Similarly, we can implement and analyze **HessianBoost** and **∞D Coordinate Descent**.

On a more practical side, a Data Scientist using our algorithms might want to know which of the predictors is the most useful. We can come up with **importance measures** for each of the predictors given by by a sum of the coefficient of that variable going into each of the weak-learners. This is very easy to do and is in line with what is done with trees [Breiman, 2017]. Similarly one could plot **partial dependence** plots similar to gradient boosting as in [Friedman, 2001]. To make the fitting process faster and more competitive one could try and use **stochasticity** in solving the subproblem (6.1), that is we use a subset of the data to find the next best predictor.

On the **theoretical side**, there is much to be done. While **Stagewise** algorithms are very appealing, their approximation to the solution of our main ℓ_1 regularized problem is unknown. While Stagewise methods, with their stable paths and generalization properties do not need justification via the lasso, an exact connection in cases like ours, where the number of predictors is infinity, should be illuminating. Similarly nothing is known about the effect of **Alternating Descent** on the solution path, while its utility is clearly seen in practice.

Our formulation of the one hidden layer neural network as a convex problem, should allow us to better understand it. Especially, for a given value of bound on the norm of fit, we must be able to establish the complexity of the fit via a **degrees of freedom** estimate. The degrees of freedom for various kinds of lasso are well known [Tibshirani et al., 2011], it must be possible to extend it to our example.

Bibliography

[Bach, 2014] Bach, F. (2014). Breaking the curse of dimensionality with convex neural networks. *arXiv preprint arXiv:1412.8690*.

[Bengio et al., 2006] Bengio, Y., Le Roux, N., Vincent, P., Delalleau, O., and Marcotte, P. (2006). Convex neural networks. *Advances in neural information processing systems*, 18:123.

[Bezanson et al., 2017] Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98.

[Boyd et al., 2016] Boyd, N., Hastie, T., Boyd, S., Recht, B., and Jordan, M. (2016). Saturating splines and feature selection. *arXiv preprint arXiv:1609.06764*.

[Boyd et al., 2015] Boyd, N., Schiebinger, G., and Recht, B. (2015). The alternating descent conditional gradient method for sparse inverse problems. In *Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP), 2015 IEEE 6th International Workshop on*, pages 57–60. IEEE.

[Boyd and Vandenberghe, 2004] Boyd, S. and Vandenberghe, L. (2004). *Convex optimization*. Cambridge university press.

[Breiman, 1993] Breiman, L. (1993). Hinging hyperplanes for regression, classification, and function approximation. *Information Theory, IEEE Transactions on*, 39(3):999–1013.

[Breiman, 2017] Breiman, L. (2017). *Classification and regression trees*. Routledge.

[Bubeck, 2015] Bubeck, S. (2015). Convex optimization: Algorithms and complexity. *Foundations and Trends® in Machine Learning*, 8(3-4):231–357.

[Chen et al., 2001] Chen, S. S., Donoho, D. L., and Saunders, M. A. (2001). Atomic decomposition by basis pursuit. *SIAM review*, 43(1):129–159.

[Cho and Saul, 2009] Cho, Y. and Saul, L. K. (2009). Kernel methods for deep learning. In *Advances in neural information processing systems*, pages 342–350.

[Efron et al., 2004] Efron, B., Hastie, T., Johnstone, I., Tibshirani, R., et al. (2004). Least angle regression. *The Annals of statistics*, 32(2):407–499.

[Elad et al., 2006] Elad, M., Milanfar, P., and Rubinstein, R. (2006). Analysis versus synthesis in signal priors. *European Signal Processing Conference*, 23:947–968.

[Freund and Schapire, 1997] Freund, Y. and Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139.

[Friedman et al., 2007] Friedman, J., Hastie, T., Höfling, H., Tibshirani, R., et al. (2007). Pathwise coordinate optimization. *The annals of applied statistics*, 1(2):302–332.

[Friedman et al., 2010] Friedman, J., Hastie, T., and Tibshirani, R. (2010). Regularization paths for generalized linear models via coordinate descent. *Journal of statistical software*, 33(1):1.

[Friedman, 2001] Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232.

[Friedman, 2012] Friedman, J. H. (2012). Fast sparse regression and classification. *International Journal of Forecasting*, 28(3):722–738.

[Friedman and Stuetzle, 1981] Friedman, J. H. and Stuetzle, W. (1981). Projection pursuit regression. *Journal of the American statistical Association*, 76(376):817–823.

[Friedman and Tukey, 1974] Friedman, J. H. and Tukey, J. W. (1974). A projection pursuit algorithm for exploratory data analysis. *IEEE Transactions on computers*, 100(9):881–890.

[Gill et al., 1981] Gill, P. E., Murray, W., and Wright, M. H. (1981). Practical optimization.

[Hastie et al., 2001] Hastie, T., Tibshirani, R., and Friedman, J. (2001). *The elements of statistical learning*, volume 1. Springer series in statistics New York, NY, USA:.

[Jaggi, 2013] Jaggi, M. (2013). Revisiting frank-wolfe: Projection-free sparse convex optimization. In *ICML (1)*, pages 427–435.

[Kiefer and Wolfowitz, 1956] Kiefer, J. and Wolfowitz, J. (1956). Consistency of the maximum likelihood estimator in the presence of infinitely many incidental parameters. *The Annals of Mathematical Statistics*, pages 887–906.

[Kim et al., 2009] Kim, S.-J., Koh, K., Boyd, S., and Gorinevsky, D. (2009). \ell_1 trend filtering. *SIAM review*, 51(2):339–360.

[Laird, 1978] Laird, N. (1978). Nonparametric maximum likelihood estimation of a mixing distribution. *Journal of the American Statistical Association*, 73(364):805–811.

[Le Roux and Bengio, 2007] Le Roux, N. and Bengio, Y. (2007). Continuous neural networks. In *Artificial Intelligence and Statistics*, pages 404–411.

[Leshno et al., 1993] Leshno, M., Lin, V. Y., Pinkus, A., and Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867.

[Mammen et al., 1997] Mammen, E., van de Geer, S., et al. (1997). Locally adaptive regression splines. *The Annals of Statistics*, 25(1):387–413.

[Mei et al., 2018] Mei, S., Montanari, A., and Nguyen, P.-M. (2018). A mean field view of the landscape of two-layer neural networks. *Proceedings of the National Academy of Sciences*, 115(33):E7665–E7671.

[Nair and Hinton, 2010] Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814.

[Rahimi and Recht, 2008] Rahimi, A. and Recht, B. (2008). Random features for large-scale kernel machines. In *Advances in neural information processing systems*, pages 1177–1184.

[Rockafellar, 1997] Rockafellar, R. T. (1997). Convex analysis. princeton landmarks in mathematics.

[Rosset et al., 2007] Rosset, S., Swirszcz, G., Srebro, N., and Zhu, J. (2007). 1 regularization in infinite dimensional feature spaces. In *International Conference on Computational Learning Theory*, pages 544–558. Springer.

[Rosset et al., 2004] Rosset, S., Zhu, J., and Hastie, T. (2004). Boosting as a regularized path to a maximum margin classifier. *Journal of Machine Learning Research*, 5(Aug):941–973.

[Sadhanala and Tibshirani, 2017] Sadhanala, V. and Tibshirani, R. J. (2017). Additive models with trend filtering. *arXiv preprint arXiv:1702.05037*.

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

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

[Tibshirani, 2014] Tibshirani, R. J. (2014). Adaptive piecewise polynomial estimation via trend filtering. *The Annals of Statistics*, 42(1):285–323.

[Tibshirani, 2015] Tibshirani, R. J. (2015). A general framework for fast stagewise algorithms. *Journal of Machine Learning Research*, 16:2543–2588.

[Tibshirani et al., 2011] Tibshirani, R. J., Taylor, J., et al. (2011). The solution path of the generalized lasso. *The Annals of Statistics*, 39(3):1335–1371.

[Tseng, 2001] Tseng, P. (2001). Convergence of a block coordinate descent method for nondifferentiable minimization. *Journal of optimization theory and applications*, 109(3):475–494.

[Wei et al., 2018] Wei, C., Lee, J. D., Liu, Q., and Ma, T. (2018). On the margin theory of feedforward neural networks. *arXiv preprint arXiv:1810.05369*.

[Zhao and Yu, 2007] Zhao, P. and Yu, B. (2007). Stagewise lasso. *Journal of Machine Learning Research*, 8(Dec):2701–2726.

[Zhou and Feng, 2017] Zhou, Z.-H. and Feng, J. (2017). Deep forest: Towards an alternative to deep neural networks. *arXiv preprint arXiv:1702.08835*.