2017

Bilinear and parallel prediction methods

Charles Hubbard
Iowa State University

Recommended Citation

Hubbard, Charles, "Bilinear and parallel prediction methods" (2017). *Graduate Theses and Dissertations*. 16147.
https://lib.dr.iastate.edu/etd/16147

Bilinear and parallel prediction methods

by

Charles Glenn Hubbard

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering (Software Systems)

Program of Study Committee:
Chinmay Hegde, Major Professor
Daji Qiao
Chao Hu

The student author and the program of study committee are solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2017

DEDICATION

To mom and dad.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

ACKNOWLEDGEMENTS

Dr. Chinmay Hegde: without you this would not have been possible. Your kindness and patience are awe-inspiring and I will be forever be grateful for all you have taught me. Thank you.

Dr. Chao Hu and Dr. Daji Qiao: Thank you both for serving on my committee and taking the time to read this thesis. Additionally, Dr. Hu, you were a wonderful mentor on my first research project and Dr. Qiao, thank you for the endless guidance you provided in my first year or graduate school.

Mom, Dad and Claire: thank you for your endless support. Also, now that I am out of school, I will call you all more often. Hopefully.

Kelly: You have kept me going throughout this process. Your support and understanding were limitless and your belief in me gave me strength I didn't know I had.

ABSTRACT

To make accurate predictions about a system one must develop a model for that system. Bilinear models are often attractive options because they allow the user to model nonlinear interactions between variables in complicated systems with (potentially) millions of variables. In this work we apply bilinear models to two separate domains and present novel models for improved prediction accuracy and novel heuristics for solving the optimization problems that arise from the use of bilinear models.

In the first system we use a bilinear model to predict the remaining useful life (RUL) of a rechargeable lithium-ion (Li-ion) battery. The approach used to solve the bilinear model leverages bilinear kernel regression to build a nonlinear mapping between the capacity feature space and the RUL state space. Specific innovations of the approach include: a general framework for robust sparse prognostics that effectively incorporates sparsity into kernel regression and implicitly compensates for errors in capacity features; and two numerical procedures for error estimation that efficiently derives optimal values of the regression model parameters.

Second, we apply a bilinear model to the *matrix completion* problem, where one seeks to recover a data matrix from a small sample of observed entries. We assume the matrix we wish to recover is low-rank (the rank of the matrix is much less than either dimension) and model it as the product of two low-rank matrices. We then adapt existing parallel solutions to this model for use on a graphics processing unit (GPU). Additionally, we introduce a novel method for inductive matrix completion on a GPU.

CHAPTER 1. INTRODUCTION

The goal of many machine learning problems is to predict an output (or a missing value) from an input vector given a (training) set of feature vectors and the known outputs. To make these predictions we are required to model the problem. Choosing a model, though, can be tricky; a simple model may give an easier-to-understand solution while a more complicated model may be capable of providing more accuracy but could also overfit to the training data. For a given problem researchers aim to find a machine learning algorithm that produces a model that is both simple and accurate. Of course, not all problems are created equal; some can be accurately modeled as linear functions of a single variable, some as linear functions of multiple variables and some can be most accurately modeled as nonlinear functions of multiple variables. A bilinear function is a nonlinear function of two variables that, when either variable of the function is held constant the function is linear in the other variable. Analogously, a bilinear map between vector spaces is map such that when either entry in the map is constant the map in linear in the other entry. In this thesis I explore two separate machine learning problems united by their use of bilinear models and alternating minimizations combined with gradient descent techniques to solve the optimization problems that define them. In both cases our problems come with real-world complications that require the use of a bilinear model. Chapter 1.1 will introduce the reader to remaining useful life prediction in lithium-ion batteries where the desire to model detect and remove errors in the training data *while we train the model* has led to the proposal of two novel algorithms for remaining useful life prediction. In Chapter 1.2 the reader will become familiar with matrix completion and inductive matrix completion. In this domain the sheer size of the data involved forces the use of bilinear models; these bilinear modes, though, give rise to parallel algorithms for completing matrices.

1.1 Prognostics for Rechargeable Lithium-ion Batteries

1.1.1 Motivation

Lithium-ion (Li-ion) battery technology has been playing a critical role in realizing wide-scale adoption of hybrid and electric vehicles and show great promise for emerging applications in smart grid and medical devices. Over the past two decades, real-time health diagnostic and prognostic techniques have been developed and deployed in battery management systems (BMSs) to monitor the health condition of a battery in operation (Plett, 2004a,b; He et al., 2013; Lee et al., 2008; Hu et al., 2012a; Xiong et al., 2014; Hu et al., 2015) and to infer, within a maintenance horizon time, the remaining useful life (RUL), i.e., when the battery is likely to fail (Saha and Goebel, 2009; Saha et al., 2009; Liu et al., 2010; Wang et al., 2013; Dickerson et al., 2015; Hu et al., 2014, 2016). Based on the voltage, current and temperature measurements acquired from the battery, these techniques estimate three performance indicators of the battery: state of charge (SOC), state of health (SOH) and state of life (SOL). Accurate estimation of these parameters provides greater transparency into the current and future health of the battery, more cost-effective maintenance strategies and minimum downtime, and opportunities for battery life extensions.

1.1.2 Relation to Prior Work

Research on life prognostics of a general engineered system was conducted with an emphasis on predicting the RUL distribution. In general, three categories of approaches have been developed that enable continuous updating of system health degradation and RUL distribution: (i) model-based approaches (Gebraeel et al., 2005; Luo et al., 2008; Gebraeel and Pan, 2008; Si et al., 2013), (ii) data-driven approaches (Si et al., 2011; Wang et al., 2008, 2012; Hu et al., 2012b; Coble and Hines, 2008; Heimes, 2008; Lu et al., 2013), and (iii) hybrid approaches (Goebel et al., 2006; Liu et al., 2012). With the advance of modern sensor systems as well as data storage and processing technologies, the data-driven approaches, which mainly rely on large volumes of sensory data with no stringent requirement on the knowledge about the underlying degradation mechanisms of the system, have recently become popular. A good review of data-driven prognostic approaches was given in (Si et al., 2011). Data-driven prognostic approaches generally require sensory data fusion and feature extraction, pattern recognition, and for life prediction, interpolation (Wang et al., 2008, 2012; Hu et al., 2012b), extrapolation (Coble and Hines, 2008), machine learning (Heimes, 2008), and so on. Research on life prognostics of Li-ion battery (or battery prognostics) was mainly conducted by researchers in the prognostics and health management (PHM) society (Saha and Goebel, 2009; Saha, Goebel, Poll, et al., 2009; Liu et al., 2010; Wang et al., 2013; Dickerson et al., 2015; Hu et al., 2014). Battery prognostics often begins by estimating the current SOH of a battery in operation based on readily available measurements (i.e., voltage, current and temperature) from the battery (Lu et al., 2013). Capacity and internal resistance are two important SOH indicators that together determine the maximum amount of energy that a fully charged battery can deliver. SOL is a prognostic metric and often used interchangeably with RUL, which refers to the available service time left before SOH of the battery degrades to an unacceptable level. RUL can be measured in either calendar time (e.g., days, weeks, and months) or charge/discharge cycles. A Bayesian framework combining the relevance vector machine (RVM), trained with sparse Bayesian learning (SBL) (Tipping, 2001), and a particle filter-based approach was proposed for prognostics of a Li-ion battery based on electrochemical impedance measurements (Saha, Goebel, Poll, et al., 2009). In order to eliminate the reliance of prognostics on impedance measurement equipment, researchers developed various model-based approaches that predict RUL by extrapolating a capacity fade model (Saha and Goebel, 2009; Liu et al., 2010; Wang et al., 2013; Dickerson et al., 2015). An integrated method for capacity estimation and RUL prediction of Li-ion battery was later developed and applied to Li-ion cells for implantable medical devices (Hu et al., 2014). The method employed the coulomb counting approach to estimate battery capacity based on the difference in the SOC values before and after partial charge/discharge. Based on the capacity estimates, a Gauss-Hermite particle filter was used to online update an empirical capacity fade model and project the updated model to an end-of-life (EOL) limit for RUL prediction. More recently, the RVM approach was leveraged to estimate battery capacity by approximating a nonlinear mapping from features (extracted from voltage and current measurements) to capacity (Hu et al., 2015; Hu et al., 2016), and RUL prediction was performed by first fitting linear models to random trajectories of capacity estimates and then extrapolating the models to an EOL limit (Hu et al., 2016).

1.1.3 Contributions

Despite significant advances in battery prognostics, research innovations are still needed to develop new approaches that can leverage large volumes of data to achieve robust RUL prediction. In particular, the goal is to perform reliable RUL prediction even in the presence of corruptions (or errors) in capacity features. In this paper, a new data-driven approach to RUL prediction is proposed and applied to a Li-ion battery used in implantable medical devices. The new approach fundamentally addresses the issue of input data noise via a new technique known as bilinear kernel regression. Specific innovations of the approach include: i) a general framework for robust sparse prognostics that effectively incorporates sparsity into kernel regression and implicitly compensates for errors in capacity features; and ii) two numerical procedures for error estimation that efficiently derive optimal values of the regression model parameters. We use 10 years' continuous cycling data on eight Li-ion prismatic cells to demonstrate the effectiveness of the proposed approach. Moreover, we compare our proposed bilinear kernel regression framework with previously existing sparse regression approaches, and demonstrate uniformly improved prediction performance. This chapters is organized as follows. Chapter 2.1 presents the fundamentals of the proposed approach. The approach is applied to a Li-ion battery used in implantable medical devices. Chapter 2.2 discusses the experimental results of this application.

1.2 Matrix Completion and Inductive Matrix Completion

1.2.1 Motivation

As businesses produce more content and offer more goods than ever before the need for trusted recommendations has never been higher. With nearly 1,000 movies and TV shows available on Neflix and Hulu and over 30 million songs available on Spotify and Apple Music it would be impossible to examine (much less sample) each option before choosing which one to interact with.

For this reason, recommendation systems have become a (in some cases the) problem of interest for large-scale content providers. Recommendation systems allow content providers to predict which items to users are most likely to enjoy and/or buy, if these recommendations are accurate, they can be of great value to the content provider. Recommendation systems fall roughly into two categories *collaborative filtering* and *content-based filtering*. Content-based filtering uses the known attributes of items(genre, year created, director, etc.) to calculate item-item similarities. Content-based filtering schemes will then recommend products to a user that either other similar users have purchased or rated highly. In contrast collaborative filtering attempts to learn users preferences solely from the ratings/purchases of other users. Collaborative filtering systems require no explicit knowledge of the items and work only on the assumption that other users' preferences can be used to predict a given users preference. Many collaborative filtering and content-based filtering methods are explored in (Schafer et al., 2007; Melville and Sindhwani, 2011). No matter the recommendation method, content providers are all trying to answer the same question: given a subset of item ratings provided by users, how best to predict future (unknown) ratings? We can reformulate that question by imagining a large ratings matrix $\mathbf{M}$ with rows representing users and columns representing movies; we are given a partially observed subset of the entries of $\mathbf{M}$ (ratings provided by users), we want to know: how can we best fill in the rest of the missing (unobserved) entries? The problem of recovering a data matrix from a small sample of its entries is called *matrix completion* and the field rose to prominence in 2009 Netflix decided offer a grand prize of $1,000,000 to anyone who could best their prediction system by over 10% (Bennett and Lanning, 2007). Matrix completion techniques were a natural solution to large-scale content recommendation problems and if it had not been before, the incentive certainly existed now.

1.2.2 Relation to Prior Work

This problem poses three central challenges. First, for a content provider such as Netflix, even the most active users can only rate a small fraction of the movies available, and therefore the matrix of *observed* ratings is extremely sparse. Without additional information, the task of recovering $\mathbf{M}$

would seem impossible. However, the recent, large body of work in matrix completion has shown that as long as the matrix $\mathbf{M}$ possess a *sufficiently low rank*, we can recover the missing entries of $\mathbf{M}$ via a convex optimization procedure (Candès and Recht, 2009; Candès and Tao, 2010; Candès and Plan, 2010; Recht, 2011).

Second, for large-scale content providers such as Netflix, Amazon, and Spotify, their popularity, and sheer amount of content available, implies that the number of users and the number of items can both be in the order of hundreds of millions (as of December 2015, Amazon had over 300 million registered users). Matrices this large (greater than a few thousand rows/columns) cause problems for typical matrix completion methods, and convex optimization approaches are not particularly suitable. To resolve this, a non-convex, incremental heuristic for matrix completion was introduced in (Recht and Ré, 2013). This method, termed as Jellyfish, achieves this speed-up by using a specific randomized version of *incremental gradient descent*, which allows data points to be processed in parallel with no fine-grained memory locking. As is the case with (Recht and Ré, 2013), many large-scale matrix completion problems require the use of *alternating minimizations* to solve non-convex optimization problems; recently a number of promising works (Jain et al., 2013; Gamarnik and Misra, 2016; Ge et al., 2017) have begun to develop theoretical guarantees for these problems.

Third, many of the canonical matrix completion methods cited above are unable to incorporate any of the so-called *side-information* we may know about the rows and columns of our matrix. In the content recommendation sense our user (row) side-information would be attributes like: age, sex and location and our item (column) attributes may be: price, genre and popularity. The problem of completing a matrix whilst taking into account the available rows/column attributes is known as *inductive matrix completion* (IMC). In (Jain and Dhillon, 2013) a theoretical foundation for IMC is provided and it is shown that, under certain conditions, the sample complexity required to solve general matrix completion problems can be substantially reduced by incorporating side information. A number of applications for IMC are explored in (Chiang et al., 2015; Natarajan and Dhillon, 2014; Yu et al., 2014a).

The application of standard approaches for matrix completion to the problem of content recommendation is not seamless. The matrix completion literature assumes that the entries of the matrix are *real-valued*; however, the ratings provided by users of content providers are almost always "quantized" to some finite set of integers (for example, Netflix ratings range from 1 to 5, while Pandora only allows a binary like/dislike system.) Treating such *categorical* ratings as if they were quantized versions of a "true" real number creates a number of issues; for example, if a user's "true" rating of a particular movie in Netflix is 7.8 but we cap the reported rating at a maximum of 5, then we have introduced a significant amount of observation noise that is unaccounted for during the recovery of the remaining ratings. To remedy this, the use of *1-Bit matrix completion*, presented in Davenport et al. (2014), has been shown to outperform contemporary matrix completion methods by intrinsically modeling the ratings as non-numeric entities

Finally, a very recent work Nisa et al. (2017) advocates a new algorithm for GPU-based matrix completion based on *cyclic coordinate descent* (CCD). Our method differs in two respects: we consider the more complicated problems of 1-bit matrix completion as well as inductive matrix completion, and our update rules involve large portions of the matrix variables (as opposed to individual coordinates). Due to space (and time) constraints, we defer a thorough comparison to future work.

1.2.3 Contributions

In this technical report, we introduce GPUFish and IMCFish, parallel computing frameworks for solving matrix completion problems for arbitrary data types. To the best of our knowledge, our proposed framework is the first to extend and massively parallelize generic matrix completion solution approaches.

GPUFish and IMCFish are modular, tunable, and leverage the massive number of multiple concurrent kernel executions possible on a modern GPU. As stylized applications, we demonstrate how to adapt GPUFish to solve the 1-bit matrix completion problem where the matrix observations are binary (Davenport et al., 2014). Our results demonstrate that we achieve a 150x speedup over existing serial algorithms, while maintaining comparable prediction accuracy. Our work demonstrates that a standard workstation equipped with a single GPU can be effectively deployed to solve very large scale matrix completion problems. We also demonstrate the use of IMCFish to solve inductive matrix completion problems. For very large IMC problems IMCFish is able to obtain competitive solutions while only having access to a fraction of the dataset at any one time.

CHAPTER 2. BILINEAR KERNEL REGRESSION

In this chapter we use bilinear kernel regression to predict the remaining useful life (RUL) of lithium-ion (Li-ion) rechargeable batteries. This work was done in collaboration with Dr. Chao Hu and John Bavlsik, both of the Department of Mechanical Engineering at Iowa State University as well as Dr. Chinmay Hegde of the Depart of Electrical and Computer Engineering at Iowa State University. This work first appeared in Hubbard et al. (2016).

2.1 Technical Approach

In this study, the capacity of a Li-ion battery cell is viewed as the SOH indicator of the cell. The cell capacity quantifies the maximum amount of charge that the cell can hold. It tends to fade slowly over time, and typically decreases 1.0% or less in a month with regular use. Given the capacity values estimated by an existing estimation algorithm, we are interested in predicting the remaining useful life (RUL) of the cell, i.e., how long the cell is expected to last before the capacity fade reaches an unacceptable level. This section is dedicated to describing the proposed data-driven approach for doing so. Chapter 2.1.1 defines the problem of data-driven prognostics considered in this study and discusses the application of kernel regression to solve this problem; Chapter 2.1.2 presents the fundamentals of a classical sparse regression technique, namely the Least Absolute Selection and Shrinkage Operator (LASSO), and discusses its application to RUL prediction when capacity estimates and RUL responses are error-free; and Chapter 2.1.3 describes the fundaments of a robust sparse regression technique, namely bilinear kernel regression, and discusses its application to RUL prediction with errors in the capacity estimates.

2.1.1 Fundamentals

Kernel regression is a non-parametric regression technique that establishes a set of identical weighted functions, called local kernels, from the training data points, and a training process is employed to adjust the weights of the kernels to achieve the best-fit line at these data points. In the context of battery prognostics, a kernel regression algorithm takes the (estimated) capacity values of a battery cell as the inputs, and produces the (predicted) RUL as the output. In this regard, kernel regression approximates the complex mapping from the capacity feature ($\mathbf{x} \in \mathbb{R}^m$) space to the RUL state ($y \in \mathbb{R}$) space.

Assume that we are given a set of training data $\{(\mathbf{x}_i, y_i\}, i = 1, 2, \ldots, n$, consisting of n samples from an arbitrary distribution D. Here, $\mathbf{x}_i$ represents the data features, each represented by an m-dimensional vector consisting of the m most-recently calculated capacity estimates. Moreover, y_i represents the (measured or true) RUL values for the corresponding capacity estimates. Our goal is to investigate a purely data-driven machine learning approach that predicts the RUL from the capacity estimates. The approach employs a nonlinear kernel regression model of the form:

$$y(\mathbf{x}) = \sum_{i=1}^{n} w_i \kappa(\mathbf{x}, \mathbf{x}_i) + w_0 \tag{2.1}$$

where $\mathbf{x}$ is a (test) feature vector, y is the predicted RUL, $w = (w_0, \ldots, w_n)^T$ represent the kernel weights, and $\kappa(\mathbf{x}, \mathbf{x}_i)$ is a suitable kernel function. The choice of kernel is somewhat flexible, but the key thing to note is that it is centered on the training point $\mathbf{x}_i$. A typical kernel used in nonlinear prediction applications is the Gaussian kernel function:

$$\kappa(\mathbf{x}, \mathbf{x}_i) = \exp\Big(- \frac{1}{r^2} ||\mathbf{x} - \mathbf{x}_i||_2^2 \Big) \tag{2.2}$$

where r is a pre-chosen parameter called the *kernel bandwidth* and $|| \cdot ||_p$ denotes the ℓ_p-norm of a vector. We use this kernel function in all our experiments below. The goal of nonlinear kernel regression is to learn the optimal model (parameterized by the weight vector $\mathbf{w}$) that provides the best prediction performance. Numerous algorithms for learning nonlinear prediction functions have been proposed in the machine learning literature, including singular value decomposition (SVD)-based approaches, stochastic gradient descent, and kernel least-squares (Trefethen and Bau III, 1997).

While kernel methods are known to provide very good prediction performance, they are often prone to overfitting to the training data and their performance can degrade on unseen test samples. Following the principle of *Occam's Razor*, machine learning algorithms for nonlinear prediction often attempt to learn a *simple* model that best explains the data. From a computational standpoint, these algorithms learn prediction models by solving a regularized problem that balances two competing objectives (training error versus model complexity). Again, numerous prediction algorithms that exploit such regularization assumptions have been developed in the literature. One approach that has been explored in detail in the PHM literature is the *Sparse Bayesian Learning* (SBL) approach (Tipping, 2001) that constructs a nonlinear regression model, known as the RVM, for online estimation of battery capacity (Hu et al., 2015, 2016). The RVM solves a Bayesian inference problem by imposing a sparse regression model on the optimal prediction weight vector, i.e., only a small subset of the coordinates of the optimal $\mathbf{w}$ are permitted to be nonzero. Tests reveal that such sparsity-based regularization methods yield better generalizability to unseen test samples, and also offer improved interpretability in terms of prediction performance.

2.1.2 RUL Prediction with the LASSO

We first propose an alternative sparsity-regularized approach for RUL prediction. Our approach is based on (now classical) optimization formulation in sparse regression called the Least Absolute Selection and Shrinkage Operator (LASSO). First, using the capacity estimates $\{\mathbf{x}_i\}$, we construct the design matrix $\mathbf{\Phi}$ of size $n \times (n+1)$, where:

$$\begin{aligned} &\phi_{i,j} = 1 \text{ for } j = 1 \text{ and} \\ &\phi_{i,j} = \kappa(\mathbf{x}_i, \mathbf{x}_{j-1}) \text{ for } j = 2, \ldots, n+1 \end{aligned} \tag{2.3}$$

Next, we arrange the corresponding RUL measurements as a response vector $\mathbf{y} = (y_0, \ldots, y_n)^T$. Finally, we define a non-negative real valued parameter λ that controls the tradeoff between the goodness of prediction fit and the sparsity of the prediction vector. Having defined these quantities, we now obtain a prediction vector by solving the convex optimization problem:

$$\hat{\mathbf{w}} = \arg\min \lambda||\mathbf{w}||_1 + ||\mathbf{y} - \Phi\mathbf{w}||_2^2 \tag{2.4}$$

The choice of the tradeoff parameter is dataset-dependent; higher values encourage greater sparsity (i.e., fewer nonzero coefficients) in the prediction vector, and vice versa. In our experiments below (see Chapter 2.2), we chose the parameters based on leave-one-out cross-validation (LOOCV).

The optimization problem in Eq. (2.4) can be solved using any of several off-the-shelf methods for convex programming, including cutting-plane methods, interior-point methods, and second-order cone programming. Since the datasets that we consider are medium-to-large scale (see Section 3.2), interior-point methods are too slow for our problem and therefore we limit our study to first-order iterative convex programming methods that only use (sub)gradient information while making progress towards the optimal solution. Specifically, in our experiments we use spectral projected gradient (SPGL1), which has been developed in the context of compressive sensing (Van Den Berg and Friedlander, 2008) for solving large-scale sparse optimization problems.

The LASSO can be viewed as a close relative of the RVM. Indeed, a Bayesian interpretation of the LASSO demonstrates that the solution to a LASSO problem is, in fact, the maximum a posteriori (MAP) estimate of the parameters w, when the prior p(w) is specified by a multi-dimensional Laplace probability density function. Several studies have shown that convex optimization methods such as the LASSO exhibit typically faster convergence (in terms of number of iterations) than Bayesian inference approaches (Roth, 2001). However, in contrast to Bayesian methods, our LASSO-based formulation does not produce a full posterior distribution of the prediction parameters. In our experiments below, we compare the LASSO with the SBL approach (Tipping, 2001).

2.1.3 RUL Prediction using Bilinear Kernel Regression

2.1.3.1 Fundamentals of Bilinear Kernel Regression

Until now, we have discussed regression approaches for prediction that (implicitly) assumes that the training samples (capacity estimates as well as RUL responses) are error-free. However, in reality, measurements (i.e., cell voltage, current and temperature) are rarely pristine. Whether due to human, instrumentation, or computation errors, it is very likely that capacity estimates are corrupted. Corruptions can occur due to noise in the measurements, owing to faulty sensor operation or variability in the power load and temperature conditions, or errors by a capacity estimation algorithm. Corruptions can also occur due to outliers, owing to sensor failure or human errors. Standard regression methods do not account for the possibility of such corruptions, and the consequence is that the inferred prediction model can be grossly incorrect, leading to unpredictable results while testing on new unseen data points. Our hypothesis is that we can build improved RUL prediction models if we explicitly capture and account for errors in the training data. We propose a unified optimization formulation for prediction of battery RUL from capacity estimates that addresses this hypothesis. First, we propose a mathematical representation of corrupted observations as follows. Suppose the (estimated) capacity measurements available to the regression method are given by:

$$\mathbf{z}_i \;=\; \mathbf{x}_i + \mathbf{u}_i \tag{2.5}$$

where $\mathbf{u}_i$ is a vector of noise values whose dimension equals the feature dimension, and whose values are generated from some probability distribution. Consequently, we use the measurements $mtbz_i$ to construct a (contaminated) kernel matrix $\mathbf{K}$ using Eq. (2.2). The relation to the ”true” kernel matrix is given by:

$$\mathbf{K} = \mathbf{\Phi} + \mathbf{E} \tag{2.6}$$

where $\mathbf{E}$ is an error matrix. The two modes of corruption that we consider are both special cases of Eq. (2.6). For the additive noise model, we assume that the capacity estimates are contaminated with independent Gaussian noise, i.e., each estimate is perturbed by a small independently chosen random variable from a normal distribution. Up to a first-order approximation, the effect of such contamination can be modeled by assuming that the entries of $\mathbf{E}$ are i.i.d. samples from a Gaussian distribution with some variance σ^2. For the outlier noise model, we assume that the capacity measurements are contaminated with sparse (but unbounded) noise, i.e., a randomly chosen fraction of the observations are arbitrary distorted. Up to a first-order approximation, the effect of such contamination can be modeled via a sparsity assumption on the error matrix.

Given the (contaminated) kernel matrix $\mathbf{K}$ and the measured (or true) RUL values $\mathbf{y}$, we solve a generalization of the optimization problem in Eq. (2.4) by jointly estimating both the optimal prediction vector as well as the error matrix:

$$(\hat{\mathbf{w}}, \hat{\mathbf{E}}) = \arg\min \lambda||\mathbf{w}||_1 + \tau||vec(\mathbf{E})||_p^p + ||\mathbf{y} - (\mathbf{K} - \mathbf{E})\mathbf{w}||_2^2 \qquad (2.7)$$

Here, the $vec()$ operator vectorizes the contents of an arbitrary matrix in column-major order. The norm parameter p is set to be either 1 or 2 depending on the noise model; the case $p = 2$ models additive noise and encourages dense estimates of the error, while the case $p = 1$ models outlier noise. As before, the parameter controls the sparsity of the final solution, while the parameter controls the norm of the aggregate errors. In theory, the solution to Eq. (2.7) will produce a sparse prediction vector that fits a kernel regression model to the ”denoised” kernel matrix . The denoising is implicit since we simultaneously remove the noise in the kernel matrix as well as estimate the prediction model. We note that unlike Eq. (2.4), the optimization problem in Eq. (2.7) is no longer convex. In particular, the squared-error loss term in Eq. (2.7) is a bilinear function of the optimization variables $\mathbf{w}$ and $\mathbf{E}$. Therefore, Eq. (2.7) is an instance of penalized bilinear regression. Variants of bilinear regression have been previously explored in the machine learning literature in (Herman and Strohmer, 2010). In particular, a similar optimization problem is proposed to develop robust versions of the LASSO that are less susceptible to outlier errors in the training data (Chen et al., 2013). To the best of our knowledge, the application of this method to battery-life prognostics has not been attempted. In our experiments below (see Chapter 2.2.2), we see that accounting for the errors in the measurements leads to improvements over the standard LASSO in all test cases, sometimes by a large amount.

2.1.3.2 Algorithm for Bilinear Kernel Regression

Since the optimization problem in Eq. (2.7) is non-convex, off-the-shelf solvers for nonlinear convex optimization cannot be directly used to solve this problem. However, due to the bilinear nature of the prediction error term in the objective function, we observe that the problem is convex, provided we fix either one of the variables ($\mathbf{w}$ or $\mathbf{E}$) and optimize over the other variable. This motivates the following, natural two-step iterative procedure based on alternating minimization:

> Step 1: Suppose we fix $\mathbf{E}$. Then, minimizing the objective function in Eq. (2.7) over all possible prediction vectors $\mathbf{w}$ reduces to a variant of the original LASSO formulation. This sub-problem can be solved using convex optimization methods such as SPGL1.
>
> Step 2: Step 1 Suppose we fix $\mathbf{w}$. Then, minimizing the objective function in Eq. (2.7) over all possible error matrices $\mathbf{E}$ reduces to an ℓ_p-regularized least squares problem. The ℓ_p-norm is convex for both $p = 1$ and $p = 2$. For $p = 1$, we can solve the sub-problem using a modification of SPGL1. For $p = 2$, the sub-problem can be reduced to ordinary penalized least-squares (also known as Tikhonov regularization (Tikhonov and Arsenin, 1977), and can be solved using standard least-squares methods such as conjugate gradients.

Algorithm 1 summarizes the overall procedure to solve the optimization problem in Eq. (2.7). The basic idea is to alternate between solving for $\mathbf{E}$ and $\mathbf{w}$. In the limit of infinitely many iterations, this procedure will converge to a local minimum of the objective function in Eq. (2.7). In practice, we can only execute a finite number of iterations. Therefore, we fix an input parameter T (representing the maximum allowable iteration count) and at the end of each iteration, we record the prediction error. The final estimates are declared by determining the iteration index that yielded the minimum objective function. The global optimality of such a method (and for non-convex optimization algorithms in general) cannot be guaranteed, but it serves as an effective heuristic. We leave as an open question the theoretical analysis of the above alternating minimization approach.

Algorithm 1 Alternating Minimizations

```
Inputs: a data training data set $\{(\mathbf{x}_i, y_i)\}, i = 1, 2, \dots, n$
Outputs: Estimated kernel prediction vector $\hat{\mathbf{w}}$
Parameters: Optimization parameters $\lambda$ and $\tau$, kernel bandwidth $r$ and the number of iterations $T$
Initialize: $\hat{\mathbf{w}}_0 \leftarrow 0,\ \hat{\mathbf{E}}_0 \leftarrow 0,\ t \leftarrow 0$
Compute: the kernel matrix $\mathbf{K}$ using Eq. (2.2)
while $t < T$ do
    Set: $\bar{\mathbf{K}} \leftarrow \mathbf{K} - \mathbf{E}$
    Solve: $\hat{\mathbf{w}}_{t+1} = \text{ arg min } \lambda||\mathbf{w}_t||_1 + ||\mathbf{y} - \Phi\mathbf{w}_t||_2^2$
    Set: $\bar{\mathbf{y}} \leftarrow \mathbf{y} - \mathbf{K}\hat{\mathbf{w}}_{t+1}$
    Solve: $\hat{\mathbf{E}}_{t+1} = \text{ arg min } \tau||vec(\mathbf{E}_t)||_p^p + ||\mathbf{y} - \mathbf{E}_t\hat{\mathbf{w}}_{t+1}||_2^2$
    Set: $\mathbf{PredictionError}_t = ||\mathbf{y} - (\mathbf{K} - \hat{\mathbf{E}}_{t+1})\hat{\mathbf{w}}_{t+1}||_2^2$
    $t \leftarrow t + 1$
end while
Find: $t^*$ that minimizes PredictionError.
Output: $\hat{\mathbf{w}}_{t^*}$
```

2.2 Experimental Results

The verification of the proposed approach was accomplished by using 10 years continuous cycling data acquired from eight Li-ion prismatic cells. This section reports the results of this verification. Chapter 2.2.1 presents the test procedure along with the cycling performance of the test cells. Chapter 2.2.2 gives the implementation details of several different methods. Chapter 2.2.3 describes the error metric used to quantify the performance of these methods in RUL prediction. The RUL prediction results are reported in Chapter 2.2.4.

2.2.1 Test Procedure and Cycling Data

Li-ion cells were constructed in hermetically sealed prismatic cases between 2002 and 2012 and subjected to full depth of discharge cycling with a nominally weekly discharge rate under 37°C (Hu et al., 2014). The cycling test was conducted with the following parameter settings: (i) the

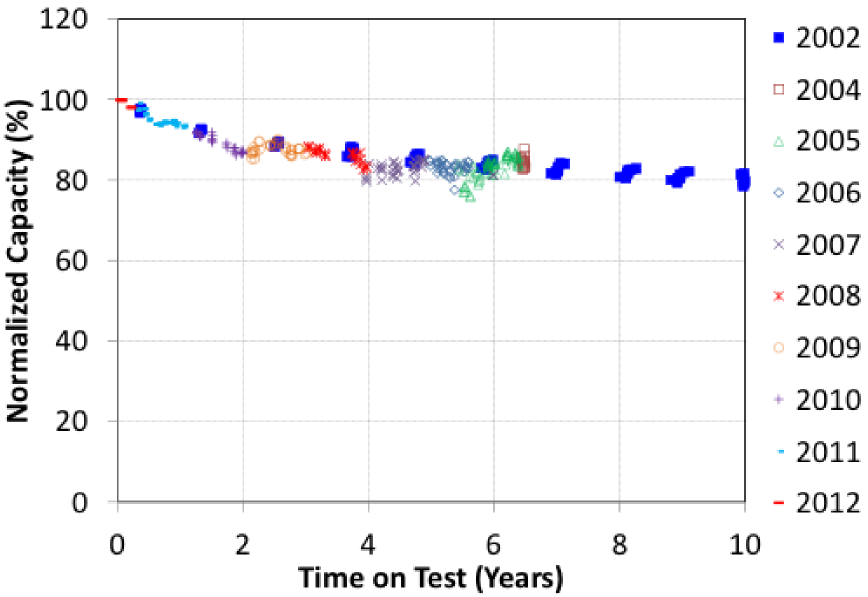


Figure 2.1: Cycling performance of cells manufactured and cycled between 2002 and 2012 Hu et al. (2014)

charge rate (I_{CC}) for the CC charge was $\frac{C}{6}$; (ii) the charge cutoff voltage (V_{max}) was 4.075 V; (iii) the time duration ($t_{CV} - t_{CC}$) of the CV charge was 30 min; and (iv) the discharge rate was $\frac{C}{150}$ or a nominally weekly discharge rate. The test attempted to simulate a use condition similar to patient use in medical applications. The weekly rate discharge capacities are plotted against the time on test in Fig. 2.1 Please note that, for confidentiality reasons, the discharge capacity of a cell in Fig. 2.1 and in the discussions thereafter is presented after being normalized by the beginning-of-life (BOL) discharge capacity of the cell. As shown in Fig. 2.1, 80% of the initial capacity is retained even after 10 years of repeated cycling at an elevated temperature, indicating exceedingly stable cell performance. The cycling data also indicate consistent performance of cells manufactured over a long time period.

2.2.2 Prognostic Data Generation and Method Implementation

In this experimental study, the cycling data from the eight 2002 cells in Fig. 2.1 were used to verify the effectiveness of the proposed approach in the RUL prediction. Each feature vector (or data point) $\mathbf{x}_i$ consists of the 3 most-recently measured capacities. To focus our discussion on RUL prediction, we did not implement capacity estimation in this study, and instead used measured capacities to construct the feature vectors. Each feature vector,$\mathbf{x}_i$, in the training data set was corrupted with additive noise,$\mathbf{n}_i$, where $\mathbf{n}_i$ is a random sample from a zero-mean normal distribution with standard deviation σ taking one of the following values: 0.0, 0.005, 0.010 and 0.015. For each σ value, all methods were tested using two 8-fold cross validation (CV) experiments: the first where the test data contained no additive noise, and the second where the test data was corrupted with additive noise from the same normal distribution as the training data. To minimize the effect of randomness in additive noise, the data generation and 8-fold cross validation were repeated 10 times.

A cell is considered to reach its' EOL when the measured discharge capacity of the cell fades to 78.5% of its initial discharge capacity (Hu et al., 2014). For any test cell whose measured capacities did not reach this EOL limit, the EOL of the cell was identified through a linear extrapolation of the capacity data from the last six charge/discharge cycles. To detect outliers in RUL prediction data, caused by spurious capacity readings input into the proposed models, a linear fit of the data was performed using the robust regression described by Holland in (Holland et al., 1977). Any residual greater than 15 median absolute deviations was removed from the prediction data and not used in the calculation of error.

The tradeoff parameters, λ in Eq. (2.4) and λ and τ in Eq. (2.7), were determined empirically and for each the value minimizing the overall root-mean-square (RMS) error (see the definition in Chapter 2.2.3) of a CV was used for all trials. For both LASSO and bilinear regression, λ was 36,000. For estimation of the error matrix, τ_{LASSO} and $\tau_{Tikhonov}$ were 26,000 and 16,000, respectively. The kernel bandwidth, r , in Eq. (2.2) was also empirically determined and was 0.05 for LASSO and bilinear regression and 0.2 for RVM.

2.2.3 Error Metric

The RUL is used as the relevant metric for determining the state of life (SOL) of Li-ion battery. We compare the prediction performance of the proposed methods (LASSO, bilinear regression with Tikhonov Regularization (our proposed algorithm with $p = 2$) to estimate errors, and bilinear regression using LASSO to estimate errors (our proposed algorithm with $p = 1$)) to that of RVM described in (Hu et al., 2015). The accuracy of a method was evaluated by using the k-fold CV. In this study, the complete feature data set $\mathbf{X}$ consists of eight mutually exclusive subsets or folds $\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_8$ that were obtained from the eight 2002 cells. In each CV trial, of the eight subsets, one was used as the test set and the other seven subsets were put together as a training set. The CV process was performed eight times, with each of the eight subsets left out exactly once as the test set. Thus, all the data points in the complete data set were used for both training and testing. Let $\mathbf{I}_l = \{i : \mathbf{x}_i \in \mathbf{X}_l\}, l = 1, 2, \ldots, 8$, denote the index set of the feature vectors that were used to construct the subset $\mathbf{X}_l$. The CV root mean square error (RMSE) is computed as the root square of the average error over all the eight CV trials, expressed as:

$$RMSE = \sqrt{\frac{1}{U} \sum_{l=1}^{8} \sum_{i \in \mathbf{I}_l} \Big(\hat{y}_{\mathbf{X} \setminus \mathbf{X}_l}(\mathbf{x}_i) - y(\mathbf{x}_i) \Big)} \tag{2.8}$$

where U is the number of feature vectors for the CV, $\hat{y}_{\mathbf{X} \setminus \mathbf{X}_l}$ is the predicted RUL by the method trained with the complete data set $\mathbf{X}$ excluding the subset $\mathbf{X}_l$, and $y(\mathbf{x}_i)$ is the true RUL of $\mathbf{x}_i$. The error formula in Eq. (2.8) indicates that all the U feature vectors in the complete data set X are used for both training and testing, and each feature vector is used for testing exactly once and for training seven times.

2.2.4 RUL Prediction Results

To determine the accuracy of a given prediction method, the errors across all cells of the CV were aggregated and a single RMSE value was calculated using Eq. (2.8). Table 2.1 summarizes the accuracy of all prediction methods with variable amounts of noise in the training data and test data. With uncorrupted test data, RVM was outperformed by all prediction methods proposed in this paper; LASSO without error estimation was outperformed by both methods incorporating error estimation. Moreover, error modeling with Tikhonov regularization outperformed error modeling with LASSO for the three of the four test cases. When test data is corrupted with small amounts of noise all models provide predictions similar to the error-free predictions in Table 2.1; greater levels of noise in the test data overwhelms the models ability to make accurate predictions. When predictions were performed with additive noise in the test set σ is the same for both training and test data.

Table 2.1: A summary of the prediction RMSE's (Eq. (2.8))

Prediction Method	Noise in training data (σ)				Noise in training and test data (σ)		
	0.000	0.005	0.010	0.015	0.005	0.010	0.015
LASSO	31.24	33.05	34.72	50.42	42.33	61.53	83.67
Bilinear regression with LASSO	30.26	31.62	33.28	**46.22**	40.96	**60.16**	**82.40**
Bilinear regression with Tikhonov regularization	**29.57**	**30.92**	**32.80**	48.44	**40.57**	60.58	83.52
RVM	30.91	32.67	36.16	47.67	41.50	60.22	82.58

Figure 2.2 demonstrates an example of an estimated error matrix (displayed as a grayscale image) generated in one of the intermediate steps of bilinear regression with Tikhonov regularization. If any data point $\mathbf{x}_i$ is imbued with error, we would expect our estimation of the i^{th} row of our error matrix to be mostly non-zero (i.e. error-filled). This phenomenon is demonstrated in Fig. 2.2, and by inspection we can infer that the rows with non-zero entries correspond to erroneous data points.

Figure 2.3 shows estimates of the Cumulative Density Function (CDF) for all prediction methods. The CDF estimate was created using the aggregated absolute value of errors from an 8-cell CV. In this CV, bilinear estimation using Tikhonov regularization outperforms all prediction methods and, as expected, a larger percentage of its errors are small. For all methods explored in this paper over 60% of predictions are within 30 cycles of the true RUL.

In Table 2.2 we compare the sparsity of the prediction vectors. For RVM the percent sparsity is simply percentage of training vectors used for prediction. For the methods presented in this

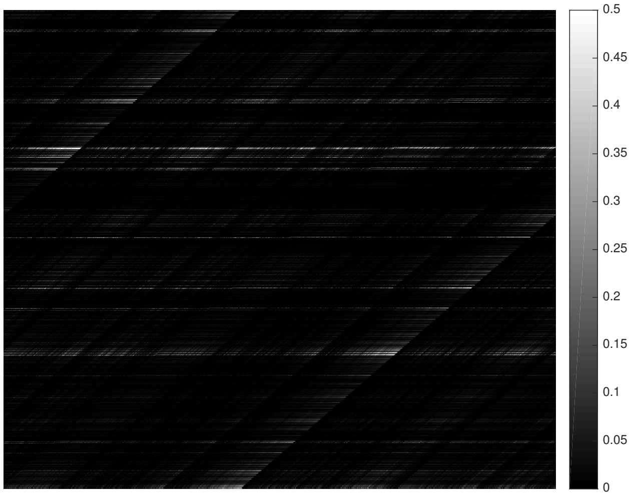


Figure 2.2: The estimated error matrix created as an intermediate step of bilinear regression with Tikhonov Regularization where $\sigma = 0.01$. Errors are reported as a percentage of the maximum value in the Gaussian kernel.

paper we determine the percent sparsity by counting the number of prediction vectors with the absolute value of their weight greater than 5% of the maximum weight. We present the median of the sparsity over an 8-fold CV where noise-free data was used for training.

Table 2.2: A comparison of the sparsity of prediction vectors.

Method	Percent Sparse
RVM	3.52
LASSO	7.78
Bilinear-LASSO	2.70
Bilinear-Tikhonov	8.82

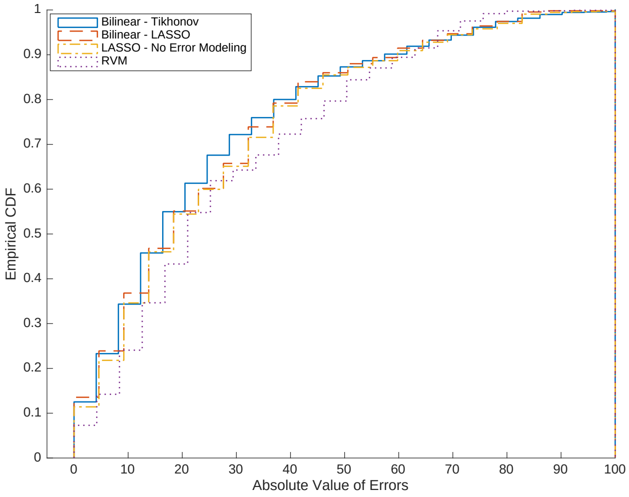


Figure 2.3: Empirical CDF vs. absolute value of error for each prediction method. Here, $\sigma = 0.01$ for the training dataset; the test data was not corrupted.

2.3 Conclusion

This chapter presents a data-driven approach to online RUL prediction of Li-ion battery by adopting bilinear kernel regression. This approach provides individual users of Li-ion battery-powered devices with estimates of the battery RUL over the whole service life. The RUL allows the users to schedule an optimal replacement near the EOL so that the devices can be used as long as possible, and at the same time, users' safety is not compromised. Our contributions to battery prognostics include the formulation of a general framework for robust sparse prognostics, and the development of two numerical procedures for efficient error estimation. Experiments with 10 years' continuous cycling data verify that the proposed approach achieves more accurate RUL prediction than existing data-driven approaches, and suggest that the proposed method is a promising methodology for the battery prognostics.

It is important to note that the experimental data in Chapter 2.2 were obtained from the eight Li-ion cells cycled with a constant discharge rate. Since the fade behavior is fairly consistent among the eight cells (see Fig. 2.1), a training data set, which carries the information about the fade behavior of 7 training cells, is likely to be capable of capturing the fade behavior of the testing cell. In non-medical applications (e.g., hybrid and electric vehicles, and consumer electronics) where harsher and more inconsistent fade scenarios are often encountered, the training data set may not fully represent the way a testing cell degrades, and in such cases, the data-driven methods discussed in this paper may produce inaccurate RUL predictions. Future work could assess the effectiveness of the proposed data-driven methods in the presence of significant cell-to-cell variation in capacity fade as well as investigate the effect of dynamic loading conditions on the accuracy in RUL prediction. Finally, we will also investigate Bayesian-inference algorithms that quantify the uncertainty in the predicted RUL estimates as a function of the noise level.

CHAPTER 3. PARALLEL HEURISTICS FOR LOW-RANK MATRIX COMPLETION

In this chapter we develop parallel, GPU-based algorithms for matrix completion and inductive matrix completion. This work done in collaboration with Dr. Chinmay Hegde of the Computer Engineering department at Iowa State University. The material from this chapter first appeared in two separate papers: Hubbard and Hegde (2016) and Hubbard and Hegde (2017).

3.1 Technical Approach

In this section we develop the problem of matrix completion and introduce an existing parallel algorithm for matrix completion. Later, we adapt that algorithm for use on a GPU and then extend our GPU-based algorithm for use in the inductive matrix completion setting.

3.1.1 Matrix Completion Fundamentals

In a matrix completion problem our goal is to complete any missing entries of a rank-r matrix $\mathbf{M}$ with n_r rows and n_c columns, given an observed subset of its entries, denoted by $\Omega \subseteq [n_r] \times [n_c]$. In a collaborative filtering setting the entries of $\mathbf{M}$ represent the interest that users have in the items available to them so the scalar M_{ij} represents the interest that user i has in item j. As with previous works in matrix completion we being by assuming our matrix $\mathbf{M}$ is at most rank-r and that r is much less than the minimum dimension of $\mathbf{M}$ (i.e. our matrix is low-rank). In the terms of an optimization problem, we seek to minimize a loss function f of our decision variable $\mathbf{X} \in \mathbb{R}^{n_r \times n_c}$ over our dataset Ω:

$$\begin{aligned} &\text{minimize} \sum_{(i,j)\ \in\Omega} f_{ij}(\mathbf{X}_{ij}), \\ &\text{s.t.}\ \ \operatorname{rank}(\mathbf{X}) \leq r. \end{aligned} \tag{3.1}$$

The optimization problem (3.1) is non-convex, due to the presence of the rank constraint on $\mathbf{X}$. The standard method in matrix completion approaches is to perform a *nuclear norm relaxation* of the rank constraint, creating a convex programming problem that can be solved by numerous methods (Davenport et al., 2014). Nuclear norm-regularized matrix recovery formulations, however, can incur a high running times (Candès and Recht, 2009; Candès and Plan, 2010; Recht and Ré, 2013) as they often require multiple SVDs of the decision variable; this problem is exacerbated when trying to perform matrix completion on large ($n_r, n_c > 5000$) matrices (Recht and Ré, 2013). To resolve this issue, we adopt the Jellyfish approach of (Recht and Ré, 2013). Jellyfish can be used to solve problems of the form:

$$\text{minimize} \quad \sum_{(i,j)\in\Omega} f_{ij}(X_{ij}) + P(\mathbf{X}), \tag{3.2}$$

where f is, again, a convex loss function and $P : \mathbb{R}^{n_r \times n_c} \to \mathbb{R}$ is a matrix regularizer that encourages low-rank solutions. While (Recht and Ré, 2013) present solutions using both the nuclear norm and the γ_2-norm as regularizers we focus on only the γ_2-norm as a regularizer (Jameson, 1987). The γ_2-norm is defined as the infimum of the matrix maximum row norms of the factors of $\mathbf{X}$, measured over all possible factorizations of $\mathbf{X}$:

$$\|\mathbf{X}\|_{\gamma_2} \;:= \inf\left\{\max\left(\|\mathbf{L}\|_{2,\infty}^2, \|\mathbf{R}\|_{2,\infty}^2\right) : \mathbf{X} = \mathbf{L}\mathbf{R}^*\right\}, \tag{3.3}$$

Here, $\|\cdot\|_{2,\infty}$ denotes the maximum row-norm of any matrix, $\mathbf{A}$:

$$\|\mathbf{A}\|_{2,\infty} := \max_j \left(\sum_k \mathbf{A}_{jk}^2\right)^{1/2}. \tag{3.4}$$

Assuming that the decision variable $\mathbf{X}$ is at most rank-r, we can rewrite it at $X = \mathbf{L}\mathbf{R}^*$, where the size of L and R are $n_r \times r$ and $n_c \times r$, respectively. Note that explicit storage of $\mathbf{X}$ requires memory capacity proportional to $n_r n_c$, which is infeasible for most matrices encountered in large-scale collaborative filtering applications. Instead, by writing our decision variable as $\mathbf{L}\mathbf{R}^*$, we only incur a memory requirement proportional to $(n_r + n_c)r$, a significant reduction (Recht and Ré, 2013).

Using our explicit factorization of **X** we can replace the γ_2-norm version of (3.2) :

$$\text{minimize} \quad \sum_{(i,j)\in\Omega} f_{ij}(\mathbf{X}) \quad \text{subject to} \quad \|\mathbf{X}\|_{\gamma_2} \leq \mathrm{B} \quad . \tag{3.5}$$

with the factored problem

$$\text{minimize} \quad \sum_{(i,j)\in\Omega} f_{ij}(\mathbf{L}\mathbf{R}^*) \quad \text{subject to} \quad \|\mathbf{L}\|_{2,\infty}^2 \leq \mathrm{B},\ \|\mathbf{R}\|_{2,\infty}^2 \leq \mathrm{B}. \tag{3.6}$$

To solve (3.6), we adopt the *incremental projected gradient descent* approach of (Recht and Ré, 2013). We alternately update **L** (resp., **R**) while keeping **R** (resp., **L**) fixed. In each iteration, the updates to **L** and **R** are given by (Recht and Ré, 2013):

$$\begin{aligned} \mathbf{L}_{i_k}^{(k+1)} &= \Pi_{\mathrm{B}}\left(\mathbf{L}_{i_k} - \alpha_k \mathcal{L}'(\mathbf{L}_{i_k}^{(k)}\mathbf{R}_{j_k}^{(k)*})\mathbf{R}_{j_k}^{(k)}\right) \\ \mathbf{R}_{i_k}^{(k+1)} &= \Pi_{\mathrm{B}}\left(\mathbf{R}_{i_k} - \alpha_k \mathcal{L}'(\mathbf{L}_{i_k}^{(k)}\mathbf{R}_{j_k}^{(k)*})\mathbf{L}_{j_k}^{(k)}\right) \end{aligned} \tag{3.7}$$

where the projection operator Π onto the constraint set in (3.6) admits the closed form expression:

$$\Pi_B(v) = \begin{cases} \frac{\sqrt{B}v}{\|v\|} & \|v\|^2 \geq \mathrm{B} \\ v & \text{otherwise} \end{cases} \quad . \tag{3.8}$$

Here $\mathbf{L}_i$ is the i^{th} row of **L** and $\mathbf{R}_j$ is the j^{th} row of **R** so $\mathbf{L}_{i_k}^{(k)}\mathbf{R}_{j_k}^{(k)*}$ is the estimated value of matrix $\mathbf{M}_{ij}$. The step size parameter in the gradient descent iteration, α_k, is a positive scalar that decreases by a constant amount at every iteration.

3.1.2 Parallel Matrix Completion

The gradient descent formulation of Eq. (3.6) has several computational advantages. Primarily, we observe that the updates performed in (3.7) operate on highly *local* portions of the matrices **L** and **R**. That is any pair $(i,j) \in \Omega$ will only read from and write to the rows $\mathbf{L}_i$ and $\mathbf{R}_j$. Given a pair of points from Ω, (i_1, j_1) and (i_2, j_2) we could perform the gradient updates for these points in parallel as long as $i_1 \neq i_2$ and $j_1 \neq j_2$; they are entirely unrelated. In the same manner, if we had two sets of points $S_1 = \{(i,j) : i \in I_1, j \in J_1\}$ and $S_2 = \{(i,j) : i \in I_2, j \in J_2\}$ with $I_1 \cap I_2 = \emptyset$ and $J_1 \cap J_2 = \emptyset$, the gradient updates for each set could be run, in principle, completely in parallel.

This is the intuition exploited in (Recht and Ré, 2013); however, they have focused on standard matrix completion as a special case, and also assume a standard multi-core computing model.

We first describe the scheme for sample ordering, called *cyclic partitioning*, developed in depth in (Recht and Ré, 2013). We introduce some terminology. Any two (row or column) index sets S_1 and S_2 are said to be overlapping if either $I_1 \cap I_2 \neq \emptyset$ or $J_1 \cap J_2 \neq \emptyset$. Therefore, if the observed data points in Ω were suitably partitioned into non-overlapping groups, then each group could be *independently* processed.

A simple way to group the data is to divide our matrix $\mathbf{M}$ into four smaller blocks. An illustration is displayed in Fig. 3.1. For consistency with (Recht and Ré, 2013) we will refer to these blocks as *chunks* and index any chunk C as $C_{a,b}$ where a and b are row and column indices of the chunk in the partitioned matrix. From Fig. 3.1, it is clear that the blocks on the diagonal of $\mathbf{M}$, $\mathbf{B_{1,1}}$ and $\mathbf{B_{2,2}}$, are non-overlapping, as are the off-diagonal blocks, $\mathbf{B_{1,2}}$ and $\mathbf{B_{2,1}}$. It is also clear that any two chunks $C_{a,b}$ and $C_{a',b'}$ are overlapping if $a = a'$ or $b = b'$.

We know that non-overlapping chunks can be safely processed in parallel; so we will define two *rounds* of chunks where $R_1 = \{C_{1,1}, C_{2,2}\}$ and $R_2 = \{C_{2,1}, C_{1,2}\}$; in this way we are free to process the chunks in each round in parallel as they are non-overlapping. If we process the rounds sequentially, no two parallel processes will ever be writing or reading from the same rows of $\mathbf{R}$ or $\mathbf{L}$ at the same time and we can eschew any *locking* delays between the different processes. Extending our example, we observe that if we have the means to process p chunks in parallel we will need to divide our matrix into p^2 chunks and divide the chunks into p rounds. Before we can determine the appropriate chunk for the data points $(i,j) \in \Omega$ we generate random permutations of the row and column indices of our matrix $\mathbf{M}$, π_{row} and π_{col} to ensure that the data points in any chunk differ between subsequent passes over that data set. In addition, we use *without-replacement* sampling to determine the order that observed samples are placed in chunks. Again, from (Recht and Ré, 2013) we place any data point $(i,j) \in \Omega$ in the chunk $C_{a,b}$ according to the following *shuffling rule*:

$$a = \left[\frac{p}{n_r}(\pi_{row}(i) - 1)\right] + 1 \text{ and } b = \left[\frac{p}{n_c}(\pi_{col}(j) - 1)\right] + 1. \tag{3.9}$$

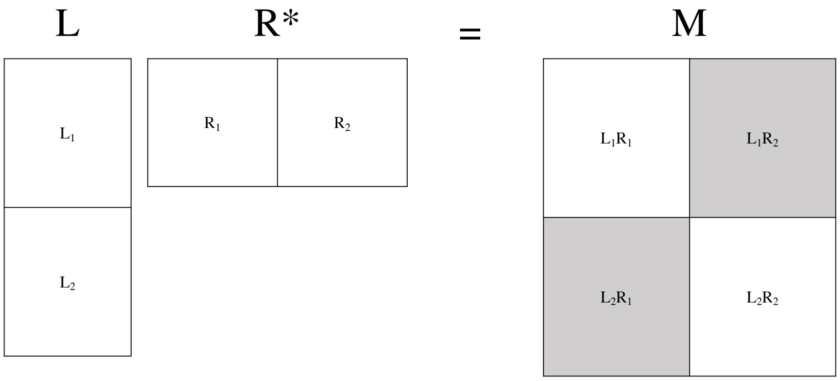


Figure 3.1: A chunking of our matrix. Non-overlapping chunks are of the same color and are grouped into rounds. Here we have 4 chunks in 2 rounds.

With the entire data set placed into chunks we are ready to perform our parallel gradient updates.

3.1.3 GPUFish

To begin, we provide a high level description of the organization of a GPU. Each process instantiated on the GPU is known as a *kernel*. A kernel can be executed in parallel across several threads of the GPU. The programmer (or compiler) groups the parallel threads into *blocks* and the blocks into a grid of blocks. When launching a kernel on the GPU, the user controls the number of blocks to launch as well as the number of threads per block to launch. Each thread launched by the kernel executes an instance of that kernel. Threads in a block execute concurrently. The execution of thread blocks is performed by *streaming multiprocessors*, or SMs for short. The number of blocks that can be executed in parallel by a single SM depends on the resources used by each block, the resources available in each SM, and the number of SMs in the GPU.

We leverage this architecture for our incremental gradient descent algorithm. Suppose that we have divided Ω into p^2 chunks. We will launch a single kernel for each of the p rounds we have created. As noted above, the kernels must be launch sequentially to perform the parallel updates without fine-grained locking. Each round will contain p chunks so we will instantiate our kernel with p blocks. Each block will be responsible for performing gradient updates (3.7) for all data points (samples) in the corresponding chunk. Each block of the kernel will contain r worker threads, where r is the rank of $\mathbf{M}$. Each thread, t_k, in a given block will be responsible for updating $\mathbf{L}_{ik}$ and $\mathbf{R}_{jk}$, that is the k^{th} entry in the rows of $\mathbf{L}$ and $\mathbf{R}$ being updated by 3.7 according to the data point $(i, j, Y_{i,j})$. In this way, we not only perform the gradient updates for a large number of data points, but also update *in parallel* the r entries of any row of $\mathbf{L}$ or $\mathbf{R}$. In contrast with (Recht and Ré, 2013), using this procedure we get an r-fold speedup *per round*.

We can further optimize running time as follows. While a given kernel (corresponding to one of the rounds) is being processed by the GPU, we simultaneous loading the data required to execute the next kernel onto the GPU. At the completion of the given we remove its data from the GPU and continue to the next round. The process of chunking our data and then performing parallel gradient updates over p kernels is known as an *epoch*. Because each epoch requires a new shuffle of our dataset, we begin each epoch by launching a separate CPU thread to compute the shuffle required for the next epoch; this extra CPU thread is executed in parallel with the GPU kernel launches being handled by the main CPU.

The procedure above is an adaptation of Jellyfish for the GPU, which we will call GPUFish. Algorithm 2 describes the overall action of a single epoch of GPUFish. Without parallelism, we find that the gradient updates of a single epoch of GPUFish takes $O(|\Omega|r)$ time as the operations in Algorithm 3 are simple and depend only on the rank, r. Additionally, permuting our data for the upcoming epoch will also be $O(|\Omega|)$ and so, in a serial setting, we find that each epoch, e, takes $O(|\Omega|r)$ time. In the parallel setting we are able to perform our gradient updates and permutations in parallel and so the time taken by each epoch will be the maximum of the the time it takes to perform our gradient updates, and the time it takes to perform the permutations of the data required for the next epoch. If we dedicate p processors to performing gradient updates and m processors to permuting the data we can perform e epochs of GPUFish in, roughly, $O(e * max(\frac{|\Omega|r}{p}, \frac{|\Omega|}{m}))$. In (Recht and Ré, 2013) the authors empirically determine the optimal values for p and m. The massive parallelism offered by a GPU not only allows GPUFish to run hundreds of gradient updates

Algorithm 2 GPUFish

```
Given: a data set $\Omega$
Permute rows and columns of $\mathbf{M}$, shuffle $\Omega$
Separate $\Omega$ into $p^2$ chunks
Round[i] $=$ $p$ chunks s.t. all chunks are non-overlapping
Transfer data for Round[1] to GPU
for i $= 1$ to $p$ in parallel do
    Launch $GPU\_Gradient\_Updates$ kernel with $p$ blocks and $r$ threads per block
    Transfer data for Round[i+1] to GPU overwriting the data from Round[i-1]
end for
```

Algorithm 3 GPU_Gradient_Updates

```
Given: $p$ chunks
for each of $p$ chunks in parallel do
    for each data point $(i, j, rating)$ in the chunk do
        apply (3.7) to $\mathbf{L}$ and $\mathbf{R}$
    end for
end for
```

in parallel but also allows us to dedicate every available CPU core to permuting our dataset. In our experiments with large data sets we found permuting the data to be the limiting factor in the computational time of GPUFish.

3.1.4 Data management

We discuss some specific schemes for managing the various observations and variables that a practical implementation of GPUFish would encounter. As mentioned above, to compute our gradient updates (3.7) we launch p blocks each with r threads; where r is the rank of our matrix. Each block is responsible for the serial processing of all the points in a single chunk and the threads allocated to each block allow us to load, update and store in memory the relevant rows $\mathbf{L}$ and $\mathbf{R}$ in a single step.

Before the first epoch, the matrices $\mathbf{L}$ and $\mathbf{R}$ are loaded into the *global memory* of the GPU where they can be accessed by all threads of the GPU. We initialize $\mathbf{L}$ and $\mathbf{R}$ with uniformly distributed random entries from $[-0.5, 0.5]$, we scale these entires by $\frac{1}{sqrt(n_r \times n_c)}$. Thread access to global memory is generally slow, so rather than make repeated calls to global memory we begin by loading the relevant rows of $\mathbf{L}$ and $\mathbf{R}$ into *shared memory* on the GPU. This memory is shared only between the threads of each block and access to it is significantly faster than global memory. After completing our computation of (3.6) from our copies of $\mathbf{L}$ and $\mathbf{R}$ in shared memory we write the our updates to $\mathbf{L}$ and $\mathbf{R}$ back to global memory.

In addition to making use of the GPU's faster shared memory, we also make use of the ability of the GPU's ability to transfer data from the while processing a kernel(s). Rather than transfer the entire permuted data set onto the GPU, at the beginning of each epoch we transfer the data needed for the first round of gradient updates and launch the kernel responsible for performing updates on the data. As this kernel processes the first round we gather the data required to process round two and load it onto the GPU. During the processing of round two we overwrite the samples corresponding to the first round with the data for the next (third round). This procedure is repeated until the epoch has finished. While providing an obvious speedup over performing all of the data transfers at once, this data management scheme also enables us to only have two rounds worth of data (approximately $\frac{2\times\|\Omega\|}{p}$ data points) on the GPU at any one. This enables GPUFish to process *very large data sets* even on memory-limited GPUs.

In Fig. 3.2 we provide a visualization of the execution of two epochs of GPUFish. We execute the first epoch without permuting the rows and columns of Ω and note the irregular sizes of the chunks. Before the second epoch begins, we shuffle the data before partitioning it into chunks. The shuffling process has spread the data points in Ω evenly across the chunks, resulting in a decreased runtime for the epoch. Each bar represents gradient updates as performed by a single chunk. Here $p = 20$ so there are 20 rounds, each containing 20 chunks; each of the 20 chunks is processed in parallel. This image was produced by NVIDIA Visual Profiler, to generate it Algorithm 2 was modified to launch p individual kernels on p streams in place of one kernel with p blocks.

3.1.5 Inductive Matrix Completion

We now define two new matrices: a matrix $\mathbf{A} \in \mathbb{R}^{n_r \times n_{d_1}}$ that contains the n_r feature vectors of size $1 \times d_1$ that describe the rows of $\mathbf{M}$ and a matrix $\mathbf{B} \in \mathbb{R}^{n_c \times n_{d_2}}$ that contains n_c information

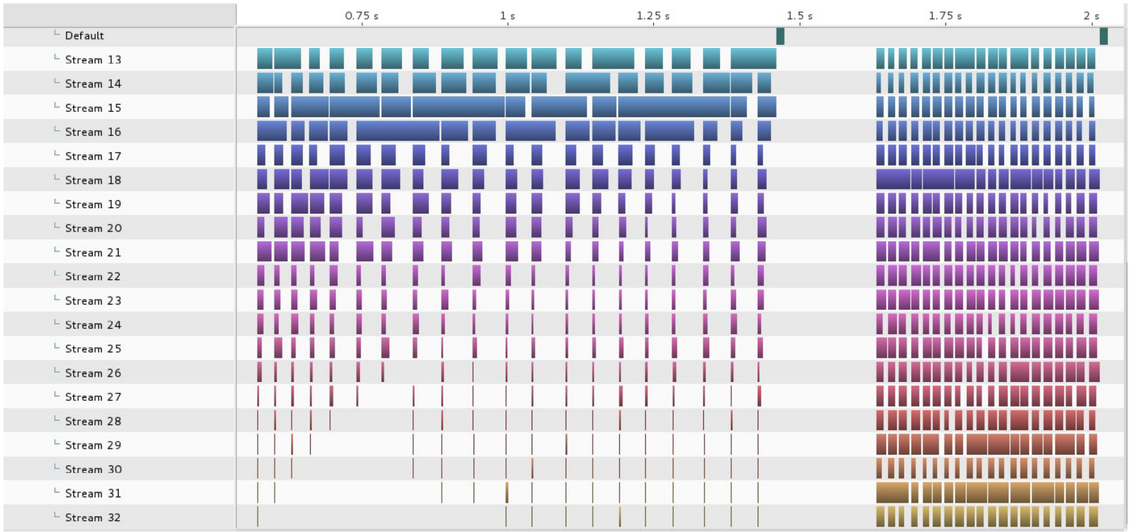


Figure 3.2: A visualization of two epochs.

vectors of size $1 \times d_2$ that describe our columns of $\mathbf{M}$ (items in the collaborative filtering setting). With $\mathbf{A}$ and $\mathbf{B}$ in hand we can obtain any entry in our observed matrix $\mathbf{M_{i,j}}$ by taking $A_i \mathbf{Z} B_j{}^*$ where $\mathbf{Z} \in \mathbb{R}^{d_1 \times d_2}$ describes the latent feature space. In the IMC setting the loss function (3.2) becomes

$$\text{minimize} \quad \sum_{(i,j)\in\Omega} \quad f_{ij}(A_i \mathbf{X} B_j^*) + P(\mathbf{X}). \tag{3.10}$$

In the same manner as our previous formulation we will assume our decision variable $\mathbf{X}$ product of two low-rank matrices $\mathbf{X} = \mathbf{L}\mathbf{R}^*$ where $\mathbf{L} \in \mathbb{R}^{d_1 \times k}$ and $\mathbf{R} \in \mathbb{R}^{d_2 \times k}$. Having factored our decision variable (3.10) becomes

$$\text{minimize} \quad \sum_{(i,j)\in\Omega} \quad f_{ij}(A_i \mathbf{L}\mathbf{R}^* B_j^*) + P(\mathbf{L}\mathbf{R}^*). \tag{3.11}$$

We again adopt the γ_2-norm as our regularizer of choice and write our factorized approximation of (3.11) as

$$\text{minimize} \quad \sum_{(i,j)\in\Omega} \quad f(\mathbf{A}_i \mathbf{L}(\mathbf{B}_j \mathbf{R})^*) \quad \text{subject to} \quad \|\mathbf{L}\|_{2,\infty}^2 \leq \mathrm{B},\ \|\mathbf{R}\|_{2,\infty}^2 \leq \mathrm{B}. \tag{3.12}$$

As we saw in (3.7) we can use incremental gradient descent to solve (3.12); our updates for each iteration are:

$$\begin{aligned}\mathbf{L}^{(k+1)} &= \Pi_{\mathrm{B}}\left(\mathbf{L} - \alpha\mathcal{L}'(\mathbf{A}_i\mathbf{L}^{(k)}\mathbf{R}^{(k)*}\mathbf{B}_j^*)\mathbf{A}_i^*\mathbf{B}_j\mathbf{R}^{(k)}\right) \\ \mathbf{R}^{(k+1)} &= \Pi_{\mathrm{B}}\left(\mathbf{R} - \alpha\mathcal{L}'(\mathbf{A}_i\mathbf{L}^{(k)}\mathbf{R}^{(k)*}\mathbf{B}_j^*)\mathbf{B}_j^*\mathbf{A}_i\mathbf{L}^{(k)}\right)\end{aligned} \tag{3.13}$$

where the projection operator Π onto the constraint set in (3.12) admits the closed form expression:

$$\Pi_B(v) = \begin{cases} \frac{\sqrt{B}v}{\|v\|} & \|v\|^2 \geq \mathrm{B} \\ v & \text{otherwise} \end{cases} . \tag{3.14}$$

While the updates described in (3.13) will minimize (3.11) they also eliminate our ability to use the biased ordering exploited by GPUFish to achieve massive parallelization. In the IMC setting, as we have seen, the entry of a matrix $\mathbf{M}_{i,j}$ is given by $A_i\mathbf{L}\mathbf{R}^*B_j^*$; each entry of $\mathbf{M}$ is a product of the entirety of $\mathbf{L}$ and the entirety of $\mathbf{R}$. In turn, the gradient updates for $\mathbf{L}$ and $\mathbf{R}$ affect all of $\mathbf{L}$ and all $\mathbf{R}$. In contrast with IMC, the updates in (3.7) only affected local portions of our decision variables, this allowed us to use a biased ordering to introduce parallelism.

3.1.6 IMCFish

To perform inductive matrix completion in parallel we first introduce the concept of striping. Given, s the number of stripes of a vector, the q^{th} stripe of a vector $v \in \mathbb{R}^{1\times n}$, denoted v^q, is given by:

$$v^q = \begin{cases} 0 & \text{for v[k] } k < q * (m/s) \text{ and } k \geq (q+1) * (m/s) \\ v & \text{otherwise} \end{cases} . \tag{3.15}$$

If we multiply the transpose of our striped vector, $v^* \in \mathbb{R}^{n\times 1}$, with some other vector $t \in \mathbb{R}^{1\times m}$ we observe that the product of these two vectors, $\mathbf{Y}$, is a striped matrix and every row Y_i is zero

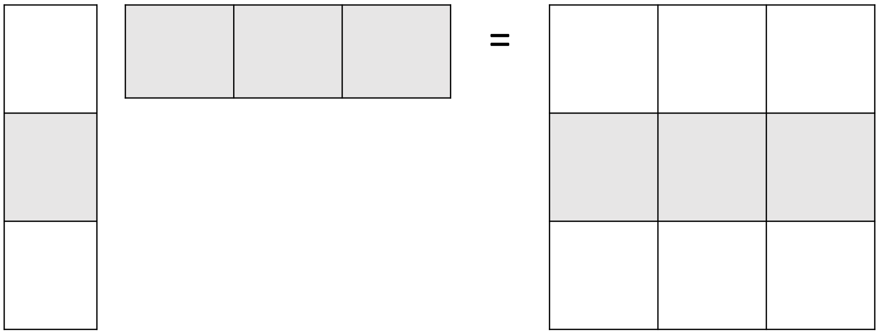


Figure 3.3: The product of multiplication with a striped vector is striped.

if the corresponding row of v^* is zero. Using this knowledge we can rewrite out updates (3.13) in terms of stripes of **A** and **B**:

$$\begin{aligned} \mathbf{L}^{q,(k+1)} = &\quad \Pi_{\mathrm{B}}\left(\mathbf{L}^{q,k} - \alpha_k f'(\mathbf{A}_i^q \mathbf{L}^{(k)} \mathbf{R}^{(k)*} \mathbf{B}_j^{*q'}) \mathbf{A}_i^{*q} \mathbf{B}_j^{q'} \mathbf{R}^{(k)}\right) \\ \mathbf{R}^{q',(k+1)} = &\quad \Pi_{\mathrm{B}}\left(\mathbf{R}^{q',k} - \alpha_k f'(\mathbf{A}_i^q \mathbf{L}^{(k)} \mathbf{R}^{(k)*} \mathbf{B}_j^{*q'}) \mathbf{B}_j^{*q'} \mathbf{A}_i^q \mathbf{L}^{(k)}\right) \end{aligned} \tag{3.16}$$

.

In this way, we are able to confine our gradient updates to *local* portions of **L** and **R** allowing us to do many updates in parallel.

Examining (3.16) we find that our updates are no longer a function of the current estimate of $\mathbf{M}_{i,j}$, given by $\mathbf{A}_i\mathbf{L}(\mathbf{B}_j\mathbf{R})^*$, but rather a fraction of our current estimate of $\mathbf{M}_{i,j}$ determined by our stripes q and q'. In practice we found this significantly hindered the convergence of (3.12) and instead computed all of the necessary estimates of $\mathbf{M}_{i,j}$ at the beginning of each epoch, where an epoch is a single pass over every point in Ω.

To run IMCFish we first choose the number of stripes to divide **A** and **B** into, say q. We retain the chunking strategy of GPUFish and divide Ω into q^2 chunks divided across q rounds. Given q stripes of **A** and **B** there are q^2 combinations of the stripes of **A** and **B**; each chunk of Ω will be responsible for performing gradient updates with a unique combination of stripes. As with GPUFish we will launch q GPU kernels each containing q blocks. The stripe of **A**, q and the stripe of **B**, q' used by a single block (chunk) is determined by the index of that block and our current round. The stripe of **A** is equal to the block index and the stripe of **B**, q' is given

Algorithm 4 IMCFish

```
$M \leftarrow$ number of users
$N \leftarrow$ number of items
$d_1 \times d_2 \leftarrow$ dimension of the latent feature space
$k \leftarrow$ rank of latent feature space
$\mathbf{M} \in \mathbb{R}^{M \times N} \leftarrow$ User-item rating information
$\Omega \leftarrow$ Set ratings visible in $\mathbf{X}$
$\mathbf{A} \in \mathbb{R}^{M \times d_1} \leftarrow$ User-features
$\mathbf{B} \in \mathbb{R}^{N \times d_2} \leftarrow$ Item-features
$\mathbf{l} \in \mathbb{R}^{d_1 \times k} \leftarrow$ random initialization
$\mathbf{r} \in \mathbb{R}^{d_2 \times k} \leftarrow$ random initialization
For any matrix $\mathbf{M}$, $\mathbf{M}_i$ is the $i^{th}$ row of $\mathbf{M}$, $\mathbf{M}^i$ is the $i^{th}$ stripe of $\mathbf{M}$

for each point $(i, j) \in \Omega$ do
    $est_{(i,j)} \leftarrow \mathbf{A}_i\mathbf{l}(\mathbf{B}_j\mathbf{r})'$
end for
Permute rows and columns of $\mathbf{M}$, shuffle $\Omega$
Separate $\Omega$ into $p^2$ chunks
Round[i] $= p$ chunks s.t. all chunks are non-overlapping
Transfer data for Round[1] to GPU
for i = 1 to $p$ do
    Launch $GPU\_Gradient\_Updates$ kernel with $p$ blocks and $k$ threads per block
    Transfer data for Round[i+1] to GPU overwriting the data from Round[i-1]
end for
```

by: $(block\ index + round)$ mod $(number\ of\ blocks)$. The full IMCFish algorithm is presented in Algorithm 4.

We note that in Algorithm 5 we only require a small fraction of our feature matrices to perform gradient updates. Say a round of chunks in IMCFish contains n_r unique rows and n_c unique columns and the feature matrices, $\mathbf{A} \in \mathbb{R}^{n_r \times d_1}$ and $\mathbf{B} \in \mathbb{R}^{n_c \times d_2}$, have been divided into q stripes each. A single gradient update only requires $\frac{d_2}{q}$ feature values from $\mathbf{A}$ and $\frac{d_2}{q}$. Given n_r rows and n_c columns we require $\frac{n_r * d_1}{q}$ features from $\mathbf{A}$ and $\frac{n_c * d_2}{q}$ features from $\mathbf{B}$ to perform all gradient updates in a given round, reducing our on-GPU memory requirement by a factor of q for each feature matrix.

At the beginning of each epoch, we compute an estimate for each point in Ω using the GPU-enabled Basic Linear Algebra Subroutines (BLAS) software, cuBLAS. The computation of a lone

Algorithm 5 GPU_Gradient_Updates

```
Given: $p$ chunks
for each $chunk$ in parallel do
    for each $(i, j, rating)$ in the chunk do
        $idx \leftarrow$ block index
        $r \leftarrow$ current round
        $s \leftarrow mod((idx + r), blocks)$
        $\mathbf{a} \leftarrow \mathbf{A}_i^{idx}$
        $\mathbf{b} \leftarrow \mathbf{B}_j^{s}$
        $\mathbf{l}^{(k+1)} \leftarrow \mathbf{l}^{(k)} - f'(est_{(i,j)})\mathbf{a}'\mathbf{b}\mathbf{r}^{(k)}$
        $\mathbf{r}^{(k+1)} \leftarrow \mathbf{r}^{(k)} - f'(est_{(i,j)})\mathbf{b}'\mathbf{a}\mathbf{l}^{(k)}$
    end for
end for
```

data point in Ω requires only a single row of $\mathbf{A}$ and a single row of $\mathbf{B}$. Given a very large data set, where the entirety of the feature matrices are unable to fit on a GPU or in main memory we are able to compute our estimates with only portions of our feature matrices. The cuBLAS computation of our estimates for points in Ω are quick and we find that the computation of our gradients is generally the limiting factor of the running time of each epoch. Without striping our expected gradient update time for each epoch is $O(|\Omega|D_1^2D_2^2r)$; if we stripe our feature matrices into q stripes our expected gradient update time per epoch becomes $O(\frac{|\Omega|D_1^2D_2^2r}{q^3})$. Striping not only allows us to perform q gradient updates in parallel it also reduces the computation required in each gradient update by a factor of q^2.

3.2 Experimental Results

We begin with the application of GPUFish to the problem of *1-bit matrix completion* (Davenport et al., 2014) and demonstrate that GPUFish can provide 100X speedups over existing serial algorithms while experiencing only minimal loss in prediction accuracy. In Chapter 3.2.2 we demonstrate the ability of IMCFish to recover random low-rank matrices at the same level as existing art while only using a fraction of the memory.

3.2.1 1-Bit Matrix Completion

As our first application, we describe an instantiation of GPUFish for solving large scale instances of the matrix completion problem where the user ratings are available in the form of binary (like/dislike) observations. We adopt the 1-bit matrix completion model of (Davenport et al., 2014). The goal is to fill in any missing entries of a rank-r matrix $\mathbf{M}$ with n_r rows and n_c columns. However, in a departure from classical matrix completion, we do not get to directly observe the entries of $\mathbf{M}$. Instead, consider any twice-differentiable function $p : \mathbb{R} \to [0, 1]$. We record observations $\mathbf{Y}$ such that:

$$Y_{i,j} = \begin{cases} +1 & \text{with probability } p(M_{i,j}), \\ -1 & \text{with probability } 1 - p(M_{i,j}), \end{cases} \quad \text{for} \quad (i,j) \in \Omega. \tag{3.17}$$

As with previous work in matrix completion, it is important that Ω is chosen *uniformly at random*. In (Davenport et al., 2014), the *Probit* and *Logit* functions are explored as natural functions to model the underlying distribution $p(\cdot)$ of the entries of $\mathbf{Y}$. In this work, we focus on the Logit function $p(x) = \frac{e^x}{1+e^x}$. To recover an estimate of $\mathbf{M}$ we can maximize the log-likelihood function of the optimization variable $\mathbf{X}$ over the set of observations Ω. Denote $\mathbb{1}_A$ as the indicator function over a Boolean condition A. Then the log-likelihood function corresponding to the Logit model is given by:

$$\mathcal{L}(\mathbf{X}) := \sum_{(i,j)\in\Omega} \left(\mathbb{1}_{Y_{i,j}=1} \log(p(X_{i,j})) + \mathbb{1}_{Y_{i,j}=-1} \log(1 - p(X_{i,j})\right). \tag{3.18}$$

The estimate of $\mathbf{M}$, therefore, is given by the solution to the constrained optimization problem[1]:

$$\widehat{\mathbf{M}} = \operatorname*{argmax}_{\mathbf{X}} \mathcal{L}(\mathbf{X}), \quad \text{rank}(\mathbf{X}) \leq r\,. \tag{3.19}$$

This, of course, is the exact problem formulation that GPUFish is designed to solve. So to perform one-bit matrix completion with GPUFish we simply replace the generic function f_{ij} that appears in Eq. (3.6) with the negative of the log-likelihood function from Eq. (3.18).

[1] To be precise, the problem formulation in (Davenport et al., 2014) also included a boundedness constraint on $\|\mathbf{M}\|_\infty$, but we omit that constraint here.

3.2.2 GPUFish Results

All experiments were performed on a Dell workstation equipped with: a 6-core Xeon E5-2620 v3 CPU, 64GB of RAM, a 256GB Class 30 SSD, and an NVIDIA GeForce GTX 1080 GPU. For our experiments, we use Linux 3.10.0-327 along with NVCC V8.0.26.

3.2.2.1 Collaborative filtering with real data

In our next batch of experiments we test the ability of GPUFish to make predictions in the collaborative filtering environment on real-world data. Specifically we make predictions about user interest in movies for the Movielens (100k , 1m and 20m) data set (Harper and Konstan, 2016). Where possible we compare our results to those produced from the code released with (Davenport et al., 2014).

We transform the user-movie ratings from the Movielens data set (integers in $[1, 5]$) to one-bit observations by subtracting the average over all ratings (approximately 3.5) from each rating and recording the sign. Our input parameters, including rank, are again determined by a grid search. Each instance of GPUFish was terminated after 20 epochs. For each Movielens data set (100k, 1M and 20M) we remove 5,000 ratings for testing purposes, and train the model with the remaining ratings. In Table 3.1 we present the percentage of one-bit ratings correctly recovered by GPUFish as a function of the original rating. We also display the overall percentage of ratings correctly recovered as well as the runtime of the algorithm.

In Table 3.2 we present the present the results of GPUFish as run on the Movielens 20M data set. Here users are allowed to rate movies on a scale from $[0.5, 1, 1.5, \ldots, 5]$.

Empirically we were able to determine the number of blocks per kernel (the number of chunks in a round) that results in the smallest run time. The results are presented in Figure 3.4. For each epoch we perform two processes in parallel: gradient updates on the GPU and the permuting and chunking Ω; the run time of each epoch is the max of the time taken by either of these two processes. Examining Figure 3.4 we see that executing GPUFish with a larger number of blocks per kernel can only decrease our runtime to the extent that it is no longer determined by the execution of

Table 3.1: A comparison between 1-bit matrix completion from (Davenport et al., 2014) and the 1-bit matrix completion implemented in GPUFish.

Original Rating	1	2	3	4	5	Overall	Runtime(s)
GPUFish: ML 100k	80%	77%	58%	71%	87%	**72%**	**0.30**
1-Bit: ML 100k	79%	73%	58%	75%	89%	**73%**	47
GPUFish: ML 1m	86%	74%	55%	75%	92%	**74%**	**1.1**
1-Bit: ML 1m	84%	76%	53%	77%	94%	**75%**	3130

Table 3.2: The results of GPUFish operating on the almost 20 million entires of the of the Movielens 20m data set. Runtime: 30 seconds.

Original Rating	0.5	1	1.5	2	2.5	3	3.5	4	4.5	5	Overall
GPUFish: ML 20m	84%	85%	89%	81%	85%	68%	63%	67%	82%	88%	74%

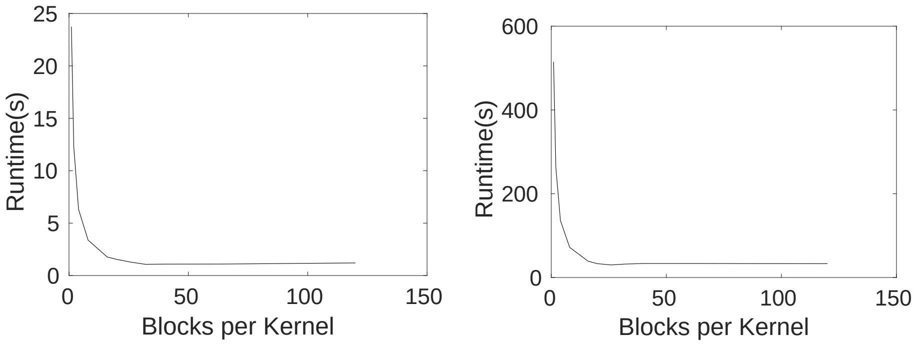
gradient updates. At approximately 30 blocks per kernel our gradient updates can be performed faster than our permutations and chunking and we no longer see a decrease in runtime.

3.2.2.2 Effect of chunking

For both the Movielens 1m and 20m data sets GPUFish was run as described in Algorithm 2 and then run with a modified algorithm where the data was permuted and chunked prior to epoch one and never again. Recall that chunking for epoch *t+1* is performed in parallel with the gradient updates for epoch t; ideally these steps would take equal amounts of time but, as shown in Table 3.3, we find that after the initial permutation of data additional permutations significantly increase the time per epoch of GPUFish, but have little effect on the accuracy of our predictions.

3.2.3 IMCFish Results

We now test IMCFish on a synthetic matrix dataset. Where possible we compare our results to those produced from the code released with (Natarajan and Dhillon, 2014).



Movielens (1m) Movielens (20m)

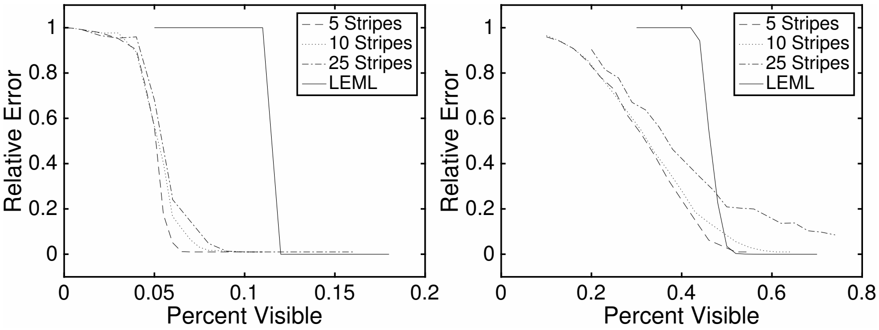
Figure 3.4: The runtime of 20 epochs of GPUFISH vs the number of blocks per kernel

Table 3.3: GPUFISH run with and without between-round chunking on the Movielens 1m and 20m data sets.

GPUFISH	Data set	Accuracy	Runtime (s)	Time per epoch (s)
w/ chunking	ML 1m	74%	1.1	0.055
w/o chunking	ML 1m	74%	0.48	0.024
w/ chunking	ML 20m	74%	30	1.5
w/o chunking	ML 20m	74%	6.5	0.33

We present phase transitions for the recovery of a random matrix, $\mathbf{M} \in \mathbb{R}^{1000\times1000}$ with rank-one and rank-ten latent feature spaces $\mathbf{Z} \in \mathbb{R}^{50\times50}$. $\mathbf{Z}$ is the product of a matrix in $\mathbb{R}^{50\times r}$ and another in $\mathbb{R}^{r\times50}$; the entries of both matrices are drawn from the standard normal distribution. To create our random feature matrices $\mathbf{A}, \mathbf{B}$, we draw from the standard normal distribution two matrices in $\mathbb{R}^{1000\times50}$. We obtain $\mathbf{M}$ by multiplying our feature matrices with $\mathbf{Z}$.

The phase transitions for IMCFISH with 5,10 and 25 stripes are presented in Figure 3.5, as are the phase transition for the LEML (Yu et al., 2014b)-based IMC algorithm detailed in (Natarajan and Dhillon, 2014). Given the output of IMCFISH $\hat{\mathbf{Z}}$, and the IMC algorithm's estimate of the latent feature space as the "ground truth" $\mathbf{Z}$, we compute the relative error of this estimate $\frac{\|\hat{\mathbf{Z}}-\mathbf{Z}\|_F^2}{\|\mathbf{Z}\|_F^2}$. In the same manner as GPUFISH, we manually tune our step size and regularization term for optimal recovery of $\mathbf{Z}$. In Fig. 3.5 the relative error of our recovery is plotted against the fraction of visible



(a) (b)

Figure 3.5: Phase transitions for the recovery of a rank one (a) and rank ten (b) matrix using IMCFISH and LEML.

entries of **M**. We note that, in the rank one case, all versions of IMCFISH, though they only use a portion of the feature matrices to perform gradient updates, are able to recover **Z** with fewer visible entires than the IMC algorithm proposed in (Natarajan and Dhillon, 2014). In the rank ten case both the five and ten striped versions of IMC perform comparably to LEML.

3.3 Conclusion

This chapter presents two algorithms for recovering low-rank matrices from a small sample of observations. In addition to being modular and tuneable both algorithms make use of the massive number of parallel operations that can be performed on a modern graphics processing unit. GPUFISH adapts JELLYFISH for the GPU and not only allows the user to solve large-scale matrix completion problems but to easily change the loss function and regularizer used to perform gradient descent. We demonstrate that GPUFISH can solve matrix completion problems orders of magnitude faster than existing art while maintaining a competitive accuracy. IMCFISH is a novel algorithm for parallel inductive matrix completion that provides accuracy on par with existing art while having a reduced memory footprint. The reduced memory footprint of IMCFISH allows the user to perform many parallel gradient updates while only need a fraction of the feature matrices on the GPU at any one time. Future work could address methods to decrease the amount of time it takes for IMCFish to converge, for example (Kingma and Ba, 2014), and the ability of IMCFish to recover the MovieLens matrix as the number of stripes in our features increases.

CHAPTER 4. CONCLUSION

In this thesis we have presented three novel bilinear and/or parallel prediction methods. The first, appearing in Chapter 2, allows the user to make accurate RUL predictions for Li-ion rechargeable batteries in the face of errors in the training data. This work has laid the foundation for battery produces to be able to be able to collect data from non-lab environments and trust that they can use it to train their algorithm without fear of noise overwhelming their model. In this way a manufacturer could even train with data collected from batteries it has distributed to customers; this would dramatically increase the amount of real-world data available. Our second two algorithms, GPUFish and IMCFish, introduce the power of GPUs to the field of matrix completion and demonstrate the usefulness of parallel computing heuristics. The massive parallelism provided by the GPU has allowed us to develop algorithms that provide comparable results to state-of-the-art algorithms while having a smaller memory footprint (IMCFish or achieving the result in a fraction of the time (GPUFish). As more and more data is collected by every content provider the importance of fast and memory efficient algorithms will only grow.

BIBLIOGRAPHY

Bennett, J. and Lanning, S. (2007). The netflix prize. In *Proc. KDD Cup and Workshop*.

Candès, E. and Plan, Y. (2010). Matrix completion with noise. *Proceedings of the IEEE*, 98(6):925–936.

Candès, E. and Tao, T. (2010). The power of convex relaxation: Near-optimal matrix completion. *IEEE Trans. Inform. Theory*, 56(5):2053–2080.

Candès, E. J. and Recht, B. (2009). Exact matrix completion via convex optimization. *Found. Comput. Math.*, 9(6):717–772.

Chen, Y., Caramanis, C., and Mannor, S. (2013). Robust sparse regression under adversarial corruption. In *International Conference on Machine Learning*, pages 774–782.

Chiang, K.-Y., Hsieh, C.-J., and Dhillon, I. (2015). Matrix completion with noisy side information. In *Adv. Neural Inf. Proc. Sys. (NIPS)*, pages 3447–3455.

Coble, J. B. and Hines, J. W. (2008). Prognostic algorithm categorization with phm challenge application. In *Prognostics and Health Management, 2008. PHM 2008. International Conference on*, pages 1–11. IEEE.

Davenport, M., Plan, Y., van den Berg, E., and Wootters, M. (2014). 1-bit matrix completion. *Information and Inference*, 3(3):189–223.

Dickerson, A., Rajamani, R., Boost, M., and Jackson, J. (2015). Determining remaining useful life for li-ion batteries. Technical report, SAE Technical Paper.

Gamarnik, D. and Misra, S. (2016). A note on alternating minimization algorithm for the matrix completion problem. *IEEE Signal Processing Letters*, 23(10):1340–1343.

Ge, R., Jin, C., and Zheng, Y. (2017). No spurious local minima in nonconvex low rank problems: A unified geometric analysis. *arXiv preprint arXiv:1704.00708*.

Gebraeel, N. and Pan, J. (2008). Prognostic degradation models for computing and updating residual life distributions in a time-varying environment. *IEEE Transactions on Reliability*, 57(4):539–550.

Gebraeel, N. Z., Lawley, M. A., Li, R., and Ryan, J. K. (2005). Residual-life distributions from component degradation signals: A bayesian approach. *IiE Transactions*, 37(6):543–557.

Goebel, K., Eklund, N., and Bonanni, P. (2006). Fusing competing prediction algorithms for prognostics. In *Aerospace Conference, 2006 IEEE*, pages 10–pp. IEEE.

Harper, F. M. and Konstan, J. (2016). The MovieLens datasets: History and context. *ACM Trans. Interactive Intelligent Systems (TiiS)*, 5(4):19.

He, W., Williard, N., Chen, C., and Pecht, M. (2013). State of charge estimation for electric vehicle batteries using unscented kalman filtering. *Microelectronics Reliability*, 53(6):840–847.

Heimes, F. O. (2008). Recurrent neural networks for remaining useful life estimation. In *Prognostics and Health Management, 2008. PHM 2008. International Conference on*, pages 1–6. IEEE.

Hu, C., Jain, G., Schmidt, C., Strief, C., and Sullivan, M. (2015). Online estimation of lithium-ion battery capacity using sparse bayesian learning. *Journal of Power Sources*, 289:105–113.

Hu, C., Jain, G., Tamirisa, P., and Gorka, T. (2014). Method for estimating capacity and predicting remaining useful life of lithium-ion battery. *Applied Energy*, 126:182–189.

Hu, C., Youn, B. D., and Chung, J. (2012a). A multiscale framework with extended kalman filter for lithium-ion battery soc and capacity estimation. *Applied Energy*, 92:694–704.

Hu, C., Youn, B. D., Wang, P., and Yoon, J. T. (2012b). Ensemble of data-driven prognostic algorithms for robust prediction of remaining useful life. *Reliability Engineering & System Safety*, 103:120–135.

Hu, X., Jiang, J., Cao, D., and Egardt, B. (2016). Battery health prognosis for electric vehicles using sample entropy and sparse bayesian predictive modeling. *IEEE Transactions on Industrial Electronics*, 63(4):2645–2656.

Hubbard, C., Bavlsik, J., Hegde, C., and Hu, C. (2016). Data-driven prognostics of lithium-ion rechargeable battery using bilinear kernel regression. In *Annual Conference of the Prognostics and Health Management Society*.

Hubbard, C. and Hegde, C. (2016). Gpufish: A parallel computing framework for matrix completion from a few observations. Technical report, Iowa State University.

Hubbard, C. and Hegde, C. (2017). Parallel computing heuristics for low-rank matrix completion. In *IEEE GlobalSIP Symposium on Accelerating Deep Learning*.

Jain, P. and Dhillon, I. (2013). Provable inductive matrix completion. *arXiv preprint arXiv:1306.0626*.

Jain, P., Netrapalli, P., and Sanghavi, S. (2013). Low-rank matrix completion using alternating minimization. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 665–674. ACM.

Jameson, G. (1987). *Summing and nuclear norms in Banach space theory*, volume 8. Cambridge University Press.

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Lee, S., Kim, J., Lee, J., and Cho, B. (2008). State-of-charge and capacity estimation of lithium-ion battery using a new open-circuit voltage versus state-of-charge. *Journal of power sources*, 185(2):1367–1373.

Liu, J., Saxena, A., Goebel, K., Saha, B., and Wang, W. (2010). An adaptive recurrent neural network for remaining useful life prediction of lithium-ion batteries. Technical report, NASA Ames Research Center.

Liu, J., Wang, W., Ma, F., Yang, Y., and Yang, C. (2012). A data-model-fusion prognostic framework for dynamic system state forecasting. *Engineering Applications of Artificial Intelligence*, 25(4):814–823.

Lu, L., Han, X., Li, J., Hua, J., and Ouyang, M. (2013). A review on the key issues for lithium-ion battery management in electric vehicles. *Journal of power sources*, 226:272–288.

Luo, J., Pattipati, K. R., Qiao, L., and Chigusa, S. (2008). Model-based prognostic techniques applied to a suspension system. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 38(5):1156–1168.

Melville, P. and Sindhwani, V. (2011). Recommender systems. In *Encyclopedia of machine learning*, pages 829–838. Springer.

Natarajan, N. and Dhillon, I. (2014). Inductive matrix completion for predicting gene–disease associations. *Bioinformatics*, 30(12):i60–i68.

Nisa, I., Sukumaran-Rajam, A., Kunchum, R., and Sadayappan, P. (2017). Parallel CCD++ on GPU for Matrix Factorization. In *Proc. General Purpose Computing Conference on GPUs*, pages 73–83.

Plett, G. L. (2004a). Extended kalman filtering for battery management systems of lipb-based hev battery packs: Part 2. modeling and identification. *Journal of Power sources*, 134(2):277–292.

Plett, G. L. (2004b). Extended kalman filtering for battery management systems of lipb-based hev battery packs: Part 3. state and parameter estimation. *Journal of Power sources*, 134(2):277–292.

Recht, B. (2011). A simpler approach to matrix completion. *J. Machine Learning Research*, 12(Dec):3413–3430.

Recht, B. and Ré, C. (2013). Parallel stochastic gradient algorithms for large-scale matrix completion. *Math. Prog. Comput.*, 5(2):201–226.

Roth, V. (2001). Sparse kernel regressors. In *International Conference on Artificial Neural Networks*, pages 339–346. Springer.

Saha, B. and Goebel, K. (2009). Modeling li-ion battery capacity depletion in a particle filtering framework. In *Proceedings of the annual conference of the prognostics and health management society*, pages 2909–2924. San Diego, CA.

Saha, B., Goebel, K., Poll, S., and Christophersen, J. (2009). Prognostics methods for battery health monitoring using a bayesian framework. *IEEE Transactions on instrumentation and measurement*, 58(2):291–296.

Schafer, J. B., Frankowski, D., Herlocker, J., and Sen, S. (2007). Collaborative filtering recommender systems. In *The adaptive web*, pages 291–324. Springer.

Si, X.-S., Wang, W., Hu, C.-H., Chen, M.-Y., and Zhou, D.-H. (2013). A wiener-process-based degradation model with a recursive filter algorithm for remaining useful life estimation. *Mechanical Systems and Signal Processing*, 35(1):219–237.

Si, X.-S., Wang, W., Hu, C.-H., and Zhou, D.-H. (2011). Remaining useful life estimation–a review on the statistical data driven approaches. *European journal of operational research*, 213(1):1–14.

Tikhonov, A. N. and Arsenin, V. Y. (1977). Solutions of ill-posed problems. vh winston & sons.

Tipping, M. E. (2001). Sparse bayesian learning and the relevance vector machine. *Journal of machine learning research*, 1(Jun):211–244.

Trefethen, L. N. and Bau III, D. (1997). *Numerical linear algebra*, volume 50. Siam.

Van Den Berg, E. and Friedlander, M. P. (2008). Probing the pareto frontier for basis pursuit solutions. *SIAM Journal on Scientific Computing*, 31(2):890–912.

Wang, D., Miao, Q., and Pecht, M. (2013). Prognostics of lithium-ion batteries based on relevance vectors and a conditional three-parameter capacity degradation model. *Journal of Power Sources*, 239:253–264.

Wang, P., Youn, B. D., and Hu, C. (2012). A generic probabilistic framework for structural health prognostics and uncertainty management. *Mechanical Systems and Signal Processing*, 28:622–637.

Wang, T., Yu, J., Siegel, D., and Lee, J. (2008). A similarity-based prognostics approach for remaining useful life estimation of engineered systems. In *Prognostics and Health Management, 2008. PHM 2008. International Conference on*, pages 1–6. IEEE.

Xiong, R., Sun, F., Chen, Z., and He, H. (2014). A data-driven multi-scale extended kalman filtering based parameter and state estimation approach of lithium-ion olymer battery in electric vehicles. *Applied Energy*, 113:463–476.

Yu, H.-F., Jain, P., Kar, P., and Dhillon, I. (2014a). Large-scale multi-label learning with missing labels. In *International Conference on Machine Learning*, pages 593–601.

Yu, H.-F., Jain, P., Kar, P., and Dhillon, I. (2014b). Large-scale multi-label learning with missing labels. In *Proc. Int. Conf. Machine Learning*, volume 32.