Nonnegative Matrix Factorization: An Empirical Analysis

Liam Geraghty Collins

Submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Engineering
Department of Electrical Engineering
Princeton University

Adviser: Yuxin Chen
Second Reader: Christopher Brinton

April 21, 2019

I hereby declare that this Independent Work report represents my own work in accordance with the University regulations.

Liam Collins

Nonnegative Matrix Factorization: An Empirical Analysis

Liam Collins

Abstract. Dimensionality-reduction and information-extraction techniques are becoming increasingly valuable as more data is collected. Yet popular techniques such as principal component analysis (PCA) and the singular value decomposition (SVD) generate factors with negative entries, severely diminishing their interpretability for nonnegative data. Nonnegative matrix factorization (NMF) is a dimensionality-reduction and information-extraction technique that aims to obtain two low-dimensional, nonnegative factors of a nonnegative dataset, where one factor is a matrix of features and the other is a matrix of weights. These features and weights reveal useful properties of the data, as their nonnegativity implies an additive features model which has insightful physical interpretations for many applications.

However, with greater potential insight comes greater computational difficulty: the nonnegativity constraints on the features and weights make computing a globally-optimal version of them a nonconvex and NP-hard problem. There exist many iterative heuristics to solve NMF that tend to converge to an effective solution in practice, but lack general performance guarantees - their behavior is still not yet thoroughly understood, especially in relation to the type of data being factored. Conversely, numerous algorithms have recently been developed with provable error bounds under certain assumptions on the data, but their practicality is questionable. Meanwhile, many initialization methods have been developed to augment the performance of NMF algorithms, yet comparative experimentation and analysis quantifying their efficacy remains insufficient in the literature.

In this paper, we comprehensively evaluate the performance of the most popular NMF algorithms and initialization techniques over varying characteristics of synthetic and real datasets in order to paint a clearer picture of when certain methods perform better and worse than others. Our analysis includes extensive background of the theory and intuition behind each technique and assessment of how this theory and intuition plays out in practice. We also evaluate how well NMF algorithms solve a particular practical problem, namely extracting latent variables from an educational dataset. Our results suggest that there is not one algorithm nor initialization technique that always performs the best, so for optimal performance, the NMF algorithm and initialization must be chosen based on the specific problem setting, and our work provides guidance for doing so.

Acknowledgements

I thank Professor Yuxin Chen for advising me on this project, and Dr. Christopher Brinton for providing helpful feedback and educational datasets.

I am eternally grateful to my parents and grandparents for their continual love and support, for providing me with the resources to attend an institution such as Princeton, and for the countless gifts they have bestowed on me through their tireless work over the years. Likewise, I am deeply thankful for all the encouragement my sister, Sarah, and my extended family have shown me while writing this thesis.

I would not have been nearly as focused, happy or motivated during this project without the compassion of my best friend, Cat. I thank her, as well as the rest of my friends in Princeton, Canton, and beyond, for their rich companionship, camaraderie, and criticism.

I am indebted to all of the professors, teachers, coaches, and advisors whom I have learned so much from over the years - especially the late Mr. Martin Badoian, whose passionate teaching and profound life lessons both inspired in me a love of math and taught me that there is always room for improvement.

Finally, I thank God for all of His abundant blessings.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Data representing many real processes, such as images, the frequencies of events, and the values of stocks, is inherently nonnegative. As larger quantities of nonnegative data are generated and collected, methods that can both accurately reduce the dimension of this data and extract meaningful information from it are becoming increasingly paramount.

Unfortunately, classical dimensionality-reduction and information-extraction techniques such as the singular value decomposition (SVD) and principal component analysis (PCA) are inadequate for nonnegative data because the low-dimensional representations they yield can have negative entries, so they cannot share the same physical interpretation as the larger dataset. This leads naturally to the problem of nonnegative matrix factorization (NMF), which seeks an accurate, low-dimensional factorization of nonnegative data and imposes a nonnegativity constraint on the factors themselves. The NMF problem is typically formulated as:

$$\begin{aligned} \min_{W\in\mathbb{R}^{p\times r}, H\in\mathbb{R}^{r\times n}} \quad & f(W,H) \\ \text{s.t.} \quad & W, H \geq 0 \end{aligned} \tag{1.1}$$

where

$$f(W,H) = \frac{1}{2}\|X - WH\|_F^2 \tag{1.2}$$

Given a nonnegative matrix $X \in \mathbb{R}_{\geq 0}^{p\times n}$, NMF seeks to compute a rank-r nonnegative factorization $WH \approx X$, where $W \in \mathbb{R}_{\geq 0}^{p\times r}$ and $H \in \mathbb{R}_{\geq 0}^{r\times n}$ are both nonnegative matrices. The *nonnegative rank* of a matrix X is defined as the largest factorization rank r such that an exact nonnegative factorization $X = WH$ exists. If the factorization rank is less than or equal to the nonnegative rank of X, then the problem of finding the exact factorization is referred to as *exact NMF*, whereas NMF itself refers to the more general problem of trying to find the best approximate factorization when the factorization rank may be greater than the nonnegative rank. In the case where a unique exact solution to the NMF problem exists, we refer to the ground-truth factors as $W^\natural$ and $H^\natural$.

In either case, not only does the factorization reduce the dimension of the representation of the dataset, but it can also extract useful information about the physical processes that the dataset represents. Specifically, the p-dimensional columns of W are the features, or basis vectors, of the data points in X, and the (i, j)-th element of H contains the weight of the i-th feature present in the j-th data point, or j-th column of X. In fact, we can think of each column j of H as an un-normalized probability distribution over the features for the j-th data point in X. Since the features in W are nonnegative, they share the same physical interpretation as the as the data in X. Moreover, because the weights H are nonnegative, they only add features to compose each data point in X, as opposed to canceling them out, which means that typically only a few basis vectors compose each data point. This sum-of-parts representation means that the optimal W and H are naturally sparse, which is beneficial for both storage and interpretability [58]. Sparsity aids interpretability because it entails that the features and weights are distinctive and representative; for example, each of the sparse bases generated thru NMF to represent a dataset of facial images often correspond to a unique, distinctive facial feature such as a nose or moustache [59].

On this note, NMF's advantages have allowed it to be applied successfully in many applications. For example, in hyperspectral imaging, the dataset X has columns containing the spectral signatures of the pixels being imaged, and the goals are to identify the materials in the image and classify each pixel by its constituent materials. Computing the nonnegative factors W and H achieves these goals because the columns of W represent the spectral signatures of each material and the columns of H represent the fractions of each material present in each pixel [7]. Elsewhere, the natural interpretations of W and H have allowed researchers to recover molecular patterns from cancer-related microarray data [19], detect brain tumors from MRI data [65], and determine the significance of topics in documents given a word-document matrix [48], among many other practical applications (see the references in [37], for example). Furthermore, the nonnegative rank of a matrix has been used to model phenomena areas including communication complexity [80], graph theory [32] and computational geometry [39].

1.1 Challenges

As powerful as the NMF solution is, computing it involves overcoming similarly substantive challenges. We detail those challenges in this section.

1.1.1 Choice of Factorization Rank

One issue with NMF is that it can be difficult to choose an appropriate factorization rank r. In some applications the problem setting clearly determines the factorization rank, but in other contexts the optimal r must be deduced. Some approaches use expert insights about the dataset or simple trial and error [37], but there also exist multiple meta learning algorithms that rigorously find the optimal r by evaluating the solutions generated for a range of r values, see for example [78] and [70]. Regardless of how it is chosen, typically $r \ll \min(p, n)$, in accordance with the dimensionality-reducing goals of NMF.

1.1.2 Identifiability

NMF also faces the problem of lacking *identifiability*: the optimal NMF (W, H) is in general not unique. Donoho and Stodden showed in their 2003 paper that NMF is identifiable when the dataset X satisfies a condition called *separability* [28], which we will discuss later. However, similar results have not been shown for general NMF. To see why, consider an orthogonal matrix $Q \in \mathbb{R}^{r\times r}$ such that $WQ \geq 0$ and $Q^{-1}H \geq 0$. Then $WH = WQQ^{-1}H$, so if (W, H) is optimal, then $(WQ, Q^{-1}H)$ is also optimal. Any permutation matrix Q may satisfy the above, but permutations of the columns of W and the rows of H do not affect their interpretations. What is more problematic is when there exists a Q that satisfies the above and is not a permutation matrix, because in this case it more fundamentally changes W and H and thus alters their interpretations.

To prevent the existence of such a Q, regularization terms are often added to the objective function (1.1) and additional priors are imposed on the factors W and H. For example, in some applications we may want to ensure that W and H are very sparse, which we can do with projections or an l_1 norm penalty [7]. Indeed, there are many variants of the general NMF problem which modify the objective function or make algorithmic adjustments in an effort to make W and/or H satisfy some additional characteristic(s) besides being nonnegative, and which are much more likely to yield an identifiable solution. See the discussion in [23] for a summary of problem variations.

One particularly notable variation of the NMF problem is to change the Frobenius norm to some other distance measure, such as the Kullback-Leibler divergence or the Itakura-Saito distance. The Frobenius norm is the most popular distance measure used for NMF because it implicitly assumes the noise in the factorization, i.e. N such that $X = WH + N$, is Gaussian, which is appropriate for many applications [37].

1.1.3 Tractability vs. Provable Accuracy

Vavasis et al. recently showed that the NMF problem 1.1-1.2 is NP-hard [72]. In general, NMF algorithms are only guaranteed to converge to stationary points, and because the NMF problem is nonconvex, these stationary points may be sub-optimal local minima. Among algorithms which are guaranteed to find an optimal factorization for the general NMF problem, the fastest executes in $O((pn)^{r^2})$ operations, due to Arora et al. [4] and Moitra [62]. This complexity is close to the lower bound of $O((pn)^{o(r)})$ operations established by Arora at al. [34] and is a polynomial runtime with r fixed. However, although r is typically small, this cost remains prohibitive for the algorithm to be used in practice.

Recently, more tractable algorithms have been presented that provably find an optimal NMF solution, but they rely on hard-to-check assumptions about the data, and may still take a prohibitively long time to solve large linear or convex programs [3, 42, 56, 12, 40, 11]. In contrast, most heuristics require only $O(pnr)$ operations and often perform well in practice despite not having provable performance guarantees [37]. These heuristics can be further sped up by choosing an effective initialization, but guarantees on the performance of different initialization techniques are similarly lacking. The lack of performance guarantees and the in-applicability of those that do exist means that a great deal remains to be understood about how NMF methods perform in practice in terms of the accuracy of the solution they yield and the time it takes them to compute it for various types of data. It is this challenge that we address in this paper, through thorough experimentation and evaluation of NMF algorithms and initialization techniques.

1.2 Related Work

Lee and Seung's seminal 1999 paper [59] applying NMF to facial recognition sparked the modern study of NMF. Since then, many researchers have studied the problem in terms of both theory and application. The algorithmic side of things, which we focus on in this paper, has experienced rapid development since Lee and Seung introduced the multiplicative updates procedure in their 1999 work [59]. Many similar iterative heuristics have been proposed in the years since, most notably alternating least squares [37], hierarchical least squares [24], alternating nonnegative least squares [60, 52, 54], and projected gradient descent [60]. Meanwhile, many initialization techniques for these NMF heuristics have been and still are being developed [21].

In 2012, a new direction in NMF algorithm research was launched by Arora's et al. paper showing that a variant of the successive projection algorithm can compute a provably accurate NMF solution under some reasonable assumptions on the data [4]. This development has led to multiple algorithms in recent years that provably compute accurate NMF solutions when the data satisfy certain conditions [3, 42, 56, 12, 40, 11]. Nevertheless, each of these algorithms and initializations requires extensive testing to determine its usefulness, because the guarantees on the provably correct algorithms do not paint a clear picture of how the algorithm will perform in practice.

Multiple comprehensive overview works have been published on NMF. The book by Cichocki et al. [23] provides a thorough exposition of algorithms that solve NMF and its many variants, and shares the results of extensive testing, much of which is application-focused. Similarly, the theses [47] and [35] and articles [7] and [75] explore the landscape of NMF algorithms, discuss variants of the NMF problem, investigate theoretical developments, and evaluate applications of the algorithms to various real-world challenges.

Despite the significant prior empirical study of NMF algorithms in these and other sources, it remains important to enhance this study. First, furthering the empirical understanding of the NMF heuristics is important simply because these heuristics lack performance guarantees, and even with the prior testing we do not yet fully understand how they behave, especially on datasets that have certain characteristics. Second, each of the prior overview works were published before the publication of the recent algorithms with provable guarantees, as well as some recent heuristics. Testing these recent algorithms beyond the small number of tests presented in the papers in which the algorithms are published is thus imperative. Similarly, the entire breadth of initialization techniques, both old and new, has yet to be thoroughly tested, necessitating such testing.

1.3 Contributions

To the best of our knowledge there does not exist any comprehensive testing and analysis of the performance of recent and older NMF algorithms and initializations across varying qualities of synthetic data like the work in this paper. By focusing primarily on synthetic data in this work, we hope to provide understanding of how well NMF algorithms will perform for certain applications based on the qualities of the data being factored, without needing to first test performance in the application setting. We test algorithm effectiveness on heavy-tailed datasets and binary datasets, for instance, to inform practitioners which NMF algorithms they should use to compute an NMF of similar datasets when they come across them in practice. We execute tests over sparsity, condition number, noise level, missing data rate, dimension, and factorization rank for the same reasons, and execute such tests over the older NMF heuristics, initialization techniques, and recent algorithms.

This is not to minimize the importance of studying applications of NMF on real data. We test initialization and algorithm performance on multiple real datasets, and study one intriguing application in depth: factoring educational datasets in the form of question-by-student binary scores. As we further detail in Chapter 4, NMF applied to educational data has the potential to be tremendously useful for educators and students alike in determining the topics or skills tested by questions and students' proficiency at these topics or skills. Surprisingly, few have studied this application so far. We further such study in this work by testing NMF algorithm performance on a new real educational dataset and executing more precise study of synthetic educational datasets.

1.4 Outline

In Chapter 2 we provide intuition for and explanation of seven of the most prominent algorithms to solve NMF. We test and analyze the performance of each of the algorithms on many types of synthetic data in Chapter 3, and investigate the application of NMF to educational data in Chapter 4. In Chapter 5 we overview eleven popular initialization techniques, then test and evaluate them on both synthetic and real datasets in Chapter 6. Chapters 7 and 8 follow a similar format, with Chapter

7 detailing seven of the most important recent NMF algorithms (both provably correct algorithms and heuristics) and Chapter 8 sharing and analyzing test results of these recent algorithms in comparison to some of the older heuristics. We close with concluding remarks in Chapter 9.

Before beginning our discussion of NMF algorithms, however, we will first use the remainder of this section to introduce notations that we will use throughout the paper.

1.5 Notations

Unless otherwise noted, we use uppercase (resp. lowercase) letters to represent matrices (resp. vectors). For a matrix A, the notation a_j denotes the j-th column of A, and a_{ij} denotes the element in the i-th row of the j-th column. To specify the i-th row of a matrix, we use the notation $A_{i,:}$ unless otherwise noted. Furthermore, $A \geq c$ for some constant c means that every element in A is $\geq c$. The notation $\mathbb{R}^{p \times r}_{\geq 0}$ refers to the space of all nonnegative p-by-n matrices. We denote the ℓ_2 norm of a vector x by $\|x\|_2$, and the transpose, Frobenius, norm, and spectral norm of a matrix A by A^T, $\|A\|_F$, $\|A\|$, respectively. $\sigma_k(A)$ denotes the k-th largest singular value of A, and $\mathrm{Tr}(A)$ is the trace of the matrix A. We use $\nabla_A f$ to denote the gradient of a function f with respect to A. $\mathrm{rank}(A)$ denotes the rank of A and $\mathrm{rank}_{\geq 0}(A)$ denotes the nonnegative rank of A. Finally, $A \circ B$ (resp. $\frac{[A]}{[B]}$) denotes the element-wise multiplication (resp. division) of two matrices A and B.

Chapter 2

Older Heuristics

As discussed in the previous chapter, NMF heuristics solve the general NMF problem in a practical number of operations (typically $O(prn)$) but have no performance guarantees. Most of these heuristics follow a *two-block coordinate descent*, also known as *alternating minimization*, scheme:

```
Algorithm 1: Two-Block Coordinate Descent
  Input: Matrix $X \in \mathbb{R}_{\geq 0}^{p \times n}$ and factorization rank $r$
  Output: Matrices $W \in \mathbb{R}_{\geq 0}^{p \times r}$ and $H \in \mathbb{R}_{\geq 0}^{r \times n}$ such that $X \approx WH$
    Generate initial matrices $W^{(0)} \in \mathbb{R}_{\geq 0}^{p \times r}$ and $H^{(0)} \in \mathbb{R}_{\geq 0}^{r \times n}$
    $t \leftarrow 1$
    while Stopping criteria not satisfied do
        $W^{(t)} \leftarrow \text{update}(X, W^{(t-1)}, H^{(t-1)})$
        $H^{(t)} \leftarrow \text{update}(X, H^{(t-1)^T}, W^{(t)^T})^T$
        $t \leftarrow t + 1$
    end while
```

This procedure computes initial estimates of W and H (for ease of notation we will drop the $\cdot^{(t)}$ superscript unless otherwise noted), then on each iteration update W and H while holding the other matrix fixed, until some stopping criteria is satisfied. There are many initialization techniques for NMF, but we will postpone the study of them until Chapter 5. Meanwhile, there are fewer stopping conditions discussed in the literature. Those that are discussed focus mainly on the relationship between consecutive iterates (e.g. [19] and [58]), the evolution of the objective function [37], and convergence to optimality conditions [60], combined with an iteration or time limit.

Indeed, it is important to know the optimality conditions for NMF, derived according to the Karush-Kuhn-Tucker criteria [37]:

$$\begin{aligned} W \geq 0 \quad \nabla_W f = (WHH^T - XH^T) \geq 0 \quad W \circ \nabla_W F = 0 \\ H \geq 0 \quad \nabla_H f = (W^T WH - W^T X) \geq 0 \quad H \circ \nabla_H F = 0 \end{aligned} \tag{2.1}$$

(W, H) is a stationary point of the objective function if and only if these conditions are satisfied [9]. The complementary slackness conditions encourage sparsity in W and H, which confers the benefits of storage and interpretability discussed in the previous chapter.

Returning to the two-block coordinate descent scheme, the update($\cdot$) function varies from algorithm to algorithm, and is the distinctive feature of each algorithm. Notably, the same update($\cdot$) function (with flipped arguments) is used to compute the next estimates of W and H within each algorithm, so our analysis of updating W also applies by symmetry to updating H and vice-versa.

The two-block structure is effective because of the convexity of each of the subblock problems. In particular, while holding H fixed, the nonnegative least squares (NNLS) subproblem $\min_{W \geq 0} \|X - WH\|_F^2$ that describes the task of updating W is convex. Although not all update($\cdot$) functions attempt to solve the NNLS subproblems exactly, they all try to find an update that will decrease the objective function (1.1), and because the subproblems are convex this generally allows for adequate performance in practice. The efficiency of two-block coordinate descent algorithms can be augmented by updating W multiple times before updating H [38]) or by greedily choosing particular elements to update [49, 81], but here for the sake of comparing the fundamental behavior of the algorithms we only test their standard versions, which update each element of W and H once per iteration.

In the remainder of this section we will discuss seven of the most prominent NMF algorithms, all but one of which follow the two-block coordinate descent scheme.

2.1 Multiplicative Updates

The Multiplicative Update (MU) procedure updates W as follows:

$$W \leftarrow W \circ \frac{[XH^T]}{[WHH^T]} \tag{2.2}$$

In practice, we add a small offset (10^{-9}) to each element in the denominator to avoid division by zero. Lee and Seung popularized this procedure by using it in their

1999 paper [59], and it has remained popular due to its simple implementation and scalability, as we will see it incurs a relatively small computational cost per iteration.

One intuition for this method comes from deriving a function $g(W, \widetilde{W})$ that is greater than or equal to $f(W)$ at every point besides (W, W), where it is strictly equal to $f(W)$, then assigning $\arg\min_W g(W, W^{(t)})$ to $W^{(t+1)}$ [67]. This technique guarantees descent:

$$f(W^{(t+1)}) \leq g(W^{(t+1)}, W^{(t)}) \leq g(W^{(t)}, W^{(t)}) = f(W^{(t)}) \tag{2.3}$$

An alternative interpretation casts MU as a gradient descent as follows [37]:

$$W \circ \frac{[XH^T]}{[WHH^T]} = W - \frac{[W]}{[WHH^T]} \circ \nabla_W F \tag{2.4}$$

Expanding on MU's relation to the gradient, we observe that

$$\begin{aligned} \frac{[XH^T]_{ik}}{[XHH^T]_{ik}} \geq 1 &\iff [XH^T]_{ik} \geq [XHH^T]_{ik} && (2.5)\\ &\iff [XHH^T]_{ik} - [XH^T]_{ik} \leq 0 && (2.6)\\ &\iff (\nabla_W F)_{ik} \leq 0 && (2.7) \end{aligned}$$

Thus, assuming $W_{ik} > 0$, the MU either increases W_{ik} if the partial derivative of the objective function with respect to it is negative, decreases it if its partial derivative is positive, or does not change it if its partial derivative is equal to zero. Each update of each $W_{ik} > 0$ then causes the objective function to decrease if that element's partial derivative is not equal to zero [37].

However, if $W_{ik} = 0$ for some (i, k), the MU does not change W_{ik}, even though its partial derivative may be negative. Because of this, the procedure is not guaranteed to converge to a stationary point. Enforcing a small positive lower bound on the W and H matrices prevents them from going to zero, which works well in practice. Still, MU typically takes many iterations to converge, as has been shown theoretically [37] and by many experiments, including ours in Chapters 3, 6 and 8.

Although MU may take many iterations to converge, one of its saving graces is that each iteration executes relatively quickly compared to other NMF algorithms. Computing the numerator of (2.2) requires $O(prn)$ operations, and the cost of computing the denominator may be reduced from $O(prn)$ to $O(\max(p, n)r^2)$ operations by computing $W(HH^T)$ instead of $(WH)H^T$ (likewise, $(W^TW)H$ instead of $W^T(WH)$ to update H). The total cost of MU is thus $O(prn)$ operations per iteration, which is the same complexity as that of other NMF heuristics, but the constant associated with this complexity is much smaller because it is due to only two matrix multiplications.

2.2 Alternating Least Squares

The Alternating Least Squares (ALS) method updates W by projecting the solution to the unconstrained least squares problem $\arg\min_U \|X - UH\|_F$ onto the nonnegative orthant [37]. Specifically, since $\arg\min_{U \in \mathbb{R}^{p\times n}} \|X - UH\|_F = (HH^T)^{-1}(XH^T)$, ALS updates W as follows:

$$W \leftarrow \mathcal{P}_{NN}[(HH^T)^{-1}(XH^T)] \tag{2.8}$$

where the projection onto the nonnegative orthant $\mathcal{P}_{NN}$ is an element-wise operation defined as:

$$\mathcal{P}_{NN}[a_{ij}] = \begin{cases} a_{ij}, & \text{if } a_{ij} \geq 0 \\ 0, & \text{if } a_{ij} < 0 \end{cases}$$

As with MU, we add a small offset to each element in $(HH^T)^{-1}$ for numerical purposes, in this case to avoid computing the inverse of an ill-conditioned matrix [7]. Also like MU, it is relatively easy to implement, and has a similarly small computational cost per iteration. Importantly, ALS typically does not converge, and may oscillate with each update. Performance may be improved by re-scaling the solution, since the updated W and H matrices are not scaled properly because of the projection [37]. In our implementation we multiply W by the constant

$$\alpha^* = \text{argmin}_{\alpha \in \mathbb{R}} \|X - \alpha WH\|_F = \frac{\langle XH^T, W\rangle}{\langle W^TW, HH^T\rangle} \tag{2.9}$$

at the start of each iteration, for accurate scaling as suggested in [37]. However, this improvement is marginal, as we observe in our experiments that the rescaled ALS method often does not converge. Nevertheless, again like MU it can still be useful as an initialization procedure because of its relatively cheap computational cost, which we also explore in Section 3.

2.3 Alternating Nonnegative Least Squares

Alternating nonnegative least squares (ANLS) algorithms attempt to solve the NNLS subproblems

$$\arg\min_{W \geq 0} \|X^T - H^TW^T\|_F^2, \quad \arg\min_{H \geq 0} \|X - WH\|_F^2 \tag{2.10}$$

on each iteration exactly, unlike ALS, and set their solutions to be the new W and H matrices. Such methods have more well-defined mathematical properties than ALS because they do not use a projection onto the nonnegative orthant. One important property is a consequence of Grippo and Siandrone's result in [46] is that any limit point of the sequence generated by the optimal solutions of each of the sub-block problems in block-coordinate descent procedures is a stationary point. The NNLS subproblems are convex, so they have at least one optimal solution. Thus, the ANLS algorithms are guaranteed to converge to a stationary point because they find the optimal solutions to the NNLS subproblems on each iteration [52].

However, the drawbacks of ANLS algorithms are that they can require many computations on each iteration to find the optimal NNLS solutions, and are often difficult to implement. Nevertheless, recent work has developed three algorithms to solve the NNLS subproblems that tend to perform well in practice: the active set [52] and block-principal pivoting [54] methods developed by Park and Kim, and the projected gradient descent algorithm proposed by Lin [60] detailed below.

2.3.1 Active Set Method

The active set method involves maintaining passive and active sets of elements of the solution matrix and exchanging elements between the sets until all the optimality conditions are satisfied [52]. We may first consider how the active set method works in the case of solving an NNLS problem over one vector, because the full matrix NNLS subproblem may be broken up into the vector NNLS subproblems as follows (solving for H this time instead of W, but similar analysis applies for updating W according to Equation 2.10):

$$\min_{H\geq 0} \|X - WH\|_F^2 \rightarrow \min_{h_1\geq 0} \|x_1 - Wh_1\|_F^2, ..., \min_{h_n\geq 0} \|x_n - Wh_n\|_F^2 \tag{2.11}$$

So, if we solve each of the n vector subproblems, we will solve the full matrix subproblem. For any $j \in [1, 2, ..., n]$, by the Karesh-Kuhn-Tucker conditions, a column vector h_j is the optimal solution to the NNLS problem arg $\min_{h_j\geq 0}\|x_j - Wh_j\|_F^2$ if and only if there exists a partitioning of the r indices into subsets F_j and G_j and a column vector $y_j = W^T(Wh_j - x_j)$ such that

$$h_{ij} = 0 \; for \; i \in F_j, \quad h_{ij} > 0 \; for \; i \in G_j \tag{2.12}$$

$$y_{ij} \geq 0 \; for \; i \in F_j, \quad y_{ij} = 0 \; for \; i \in G_j \tag{2.13}$$

where h_{ij} (resp. y_{ij}) is the element in the i-th row of the column vector h_j (resp. y_j) [52]. Here, F_j is the active set and G_j is the passive set. Kim and Park's active set method entails that each computed h_{F_j} solves the unconstrained least squares problem $\min_{h_{F_j}} \|x_j - W_{F_j} h_{F_j}\|_2$, where W_{F_j} (resp. h_{F_j}) is the submatrix of W with columns selected by F_j (resp. subvector of h_j with elements selected by F_j) and the subvectors h_{G_j} and y_{G_j} satisfy the optimality conditions.

Computational cost can be significantly reduced by computing each column of H in parallel instead of one at a time [52]. On each iteration, the active set method solves the active set indexed unconstrained least squares problems $z_{F_j} = \arg\min_{z_{F_j}} \|x_j - W_{F_j} z_{F_j}\|_2$ for each column with index j of the matrix H that is not optimal. For the unconstrained column solutions that are infeasible (that have elements less than or equal to zero), it updates the columns of H with a function of the old columns and the new column solutions z_j, and exchanges one element between the active and passive sets of each column. For the unconstrained column solutions that are feasible, it updates H with those columns, and exchanges at most one element between the active and passive sets of each of those columns which have elements that disobey the optimality conditions [52].

This version of Kim and Park's active set method that updates for each column simultaneously is overviewed in Algorithm 2. All of the computations in the algorithm can be performed on submatrices composed of sets of columns instead of iterating through individual columns, which is faster than operating on the individual columns. In particular, the active set-indexed unconstrained least squares problems can be solved simultaneously using the column grouping method proposed by Van Benthem and Keenan [6]. Their method solves the normal equations

$$W_{F_j}^T W_{F_j} z_{F_j} = W_{F_j}^T x_j \tag{2.14}$$

for all j in sets of similar passive sets F_j.

If the matrix $W^T W$ is ill-conditioned, the normal equations may not be able to be solved accurately, so the algorithm execution will be erroneous and may take a large number of iterations to fully satisfy the optimality conditions, or oscillate and never fully satisfy them at all. To account for this scenario, we impose a maximum of $5r$ on the number of iterations allowed.

Although the strategy to update columns in parallel speeds up the Active Set method significantly, it is still limited by the fact that it only exchanges at most one element between the passive and active sets for each column on every iteration, which means that the method requires a relatively large number of iterations for convergence.

Nevertheless, the algorithm is still guaranteed to converge to a stationary point in a finite number of iterations [54]. Moreover, if W has full column rank, then the NNLS subproblem is strictly convex, and the stationary point that the active set method converges to is the unique optimal solution to the subproblem.

Algorithm 2: Active set method to solve $\min_{H\geq 0} \|X - WH\|_F^2$ [52]

```
**Input:** Nonnegative matrices $X \in \mathbb{R}^{p\times n}$ and $W \in \mathbb{R}^{p\times r}$
**Output:** Nonnegative matrix $H = \text{argmin}_{H\geq 0}\|X - WH\|_F^2$
  **Initialization:**
  $H = \mathbf{0} \in \mathbb{R}^{r\times n}$, $Y = W^T X$
  $F_j = \{1, 2, ..., r\}$, $G_j = \{\}$ for all $j \in \{1, 2, ..., n\}$
  Find all column indices with not optimal elements: $E = \{j : (H_j, Y_j)$ is not optimal$\}$
  **while** $E$ is not empty **do**
      $Z(\in \mathbb{R}^{r\times n}) = \mathbf{0}$
      Solve $z_j = \text{argmin}_z \|x_j - W_{F_j} z\|_2$ for each $j \in E$ using column grouping
      Find all column indices in $E$ where $Z$ has infeasible passive set elements:
      $E_F = \{j : j \in E$ and $\exists \quad z_{ij} \leq 0$ for some $i \in F_j$
      **for all** column indices $j \in E_F$ **do**
          Find an $\alpha_j := \min\{h_{ij}/(h_{ij} - z_{ij}) : z_{ij} \leq 0, i \in F_j\}$
          Set $h_j = h_j + \alpha_j(z_j - h_j)$
          Move any index $i$ where $h_{ij} = 0$ from $F_j$ to $G_j$
      **end for**
      find all column indices in $E$ where $Z$ is feasible: $E'_F = E \setminus E_F$
      **if** $E'_F$ is not empty **then**
          For each column index $j \in E'_F$, set $h_j = z_j$ and $y_j = W^T W h_j - W^T x_j$
          Find $E_{new} = \{j : j \in E'_F$ and $\exists y_{ij} < 0$ and $i \in G_j$ for some $i\}$
          For each column index $j$ in $E_{new}$, find $\beta_j = \arg\max_i y_{ij}$ such that $i \in G_j$
          Move each index $\beta_j$ from $G_j$ to $F_j$
          Update $E = E_F \cup E_{new}$
      **end if**
  **end while**
```

2.3.2 Block Principal Pivoting

Also developed by Park and Kim [54], the block principal pivoting method follows a very similar procedure to the active set method for solving the NNLS subproblems, but has one important difference. Instead of only exchanging one element between the active and passive sets for each not optimal column after solving the unconstrained least squares problems on each iteration, the block principal pivoting method typically exchanges all of the not optimal elements of each column (whose indices are contained in the set V_j for the j-th column) between the passive and active sets on each iteration. The only instance when the block principal pivoting method does not exchange all the not optimal elements is when doing so has not caused the total number of them to decrease over the past α_j iterations, where the default value of α_j is 3. In this case, only the not-optimal element with the largest index in the column is exchanged, and this protocol is repeated until the number of not optimal elements in that column is less than the previously smallest number of not optimal elements, β_j [54]. Details of this algorithm are provided in Algorithm 3.

Algorithm 3: Block principal pivoting method to solve $\min_{H\geq 0} \|X - WH\|_F^2$ [54]

```
**Input:** Nonnegative matrices $X \in \mathbb{R}^{p\times n}$ and $W \in \mathbb{R}^{p\times r}$
**Output:** Nonnegative matrix $H = \operatorname{argmin}_{H\geq 0}\|X - WH\|_F^2$
  **Initialization:**
  $U(\in \mathbb{R}^{r\times n}) = \mathbf{0}$, $Y = W^T X$
  $F_j = \{\}$, $G_j = \{1, 2, ..., r\}$ for all $j \in \{1, 2, ..., n\}$
  $\alpha(\in \mathbb{R}^n) = 3$, $\beta(\in \mathbb{R}^n) = r + 1$
  **while** Any element in $(H, Y)$ is not optimal **do**
      Find all column indices with not optimal elements: $E = \{j : (H_j, Y_j)$ is not optimal$\}$
      Compute $V_j = \{i \in F_j : h_{ij} < 0\} \cup \{i \in G : y_{ij} < 0\}$ for all $j \in E$
      For all $j \in E$ with $| V_j | < \beta_j$, set $\beta_j =| V_j |$, $\alpha_j = 3$, and $\hat{V}_j = V_j$
      For all $j \in E$ with $| V_j | \geq \beta_j$ and $\alpha_j \geq 1$, set $\alpha_j = \alpha_j - 1$ and $\hat{V}_j = V_j$
      For all $j \in E$ with $| V_j | \geq \beta_j$ and $\alpha_j = 0$, set $\hat{V}_j = \{i : i = \max\{i \in V_j\}\}$
      **Update:**
      $F_j = (F_j \setminus \hat{V}_j) \cup (\hat{V}_j \cap G_j$ for each $j \in E$
      $G_j = (G_j \setminus \hat{V}_j) \cup (\hat{V}_j \cap F_j$ for each $j \in E$
      $h_j = \operatorname{argmin}_z \|x_j - W_{F_j} z\|_2$ for each $j \in E$ using column grouping
      $y_j = W^T W h_j - W^T x_j$ for each $j \in E$
  **end while**
```

Kim and Park give the backup single exchange rule because it is guaranteed to converge to a stationary point in a finite number of iterations (as we saw with the active set method), whereas exchanging all of the not optimal elements (the full exchange rule) on every iteration may lead to a cycle [54]. Since the full exchange rule can only be employed unsuccessfully a finite number of times before the single exchange rule is used, the block principal pivoting algorithm is guaranteed to converge to a stationary point in a finite number of iterations [54]. However, as Park and Kim have shown, in practice the full exchange rule rarely leads to a cycle, so the algorithm rarely utilizes the backup single exchange rule, and thus finds an optimal solution in much fewer iterations than exchanging only one element per iteration as in the active set method [54]. This reality, combined with the block principal pivoting method's ability to compute each column in parallel (like the active set method) means that this method is a relatively fast one for solving the NNLS subproblems. The algorithm encounters the same problems with ill-conditioning as those dealt with by the active set method, so we impose the same maximum of $5r$ on the maximum number of iterations allowed so solve each NNLS subproblem.

2.3.3 Projected Gradient Descent (within ANLS)

Lin used projected gradient descent in two separate NMF algorithms: one to directly minimize the NMF objective function (PGD), which we will discuss in Section 2.4, and another to solve the NNLS subproblems within the ANLS framework (ANLS PGD) [60]. We discuss the latter here as our final method for solving the NNLS subproblems.

Generic gradient descent attempts to find the argument A that minimizes a function f by generating the sequence

$$A^{(k)} = A^{(k-1)} - \alpha_{k-1} \nabla_A f(A^{(k-1)}) \tag{2.15}$$

where the scalar α_{k-1} is called the step size. Projected gradient descent accounts for bounds on the set of feasible arguments by projecting the RHS of the above equation onto the feasible region [60]. For NMF, in which solutions must be bounded below by zero, Lin uses the same projection operator $\mathcal{P}_{NN}$ used in ALS, which sets all negative elements to zero. To solve the NNLS subproblem with respect to H, recall

$$f(H) = \|X - WH\|_F^2 \text{ and } \nabla f(H) = W^T(WH - X) \tag{2.16}$$

We therefore have all the components necessary to use projected gradient descent to solve the NNLS subproblem with respect to H (and with respect to W using the corresponding gradient) besides the step size. To compute the step size, Lin's algorithm employs the Armijo rule [8]. For the k-th iteration, this rule sets the step size $\alpha_k = \beta^{t_k}$, where t_k is the smallest integer that satisfies

$$f(\tilde{A}^{(k)}) - f(A^{(k-1)}) \leq \sigma \nabla f(A^{(k-1)})^T (A^{(k)} - A^{(k-1)}) \tag{2.17}$$

where

$$\tilde{A}^{(k)} = \mathcal{P}[A^{(k-1)} - \alpha_k \nabla f(A^{(k-1)})] \tag{2.18}$$

Lin's full projected gradient descent algorithm is outlined in Algorithm 4, with $\mathcal{P} \equiv \mathcal{P}_{NN}$, $A \equiv H$ and f and ∇f defined as in Equation 2.16 to solve the NNLS subproblem with respect to H. Following Lin's suggestion, we use $\sigma = 0.01$ and $\beta = 0.1$ in our implementation.

Choosing α_k to satisfy Equation 2.17 ensures a sufficient decrease of the function value on each iteration. Bertsekas showed that such an $\alpha_k > 0$ always exists [8], so the function value decreases on every iteration. This sequence generated by this method has been shown to converge to a stationary point in, for example, [20]. If W is full column rank, then $f(H)$ is strictly convex, so any stationary point must be the unique global minimum.

Unlike the active set and block principal pivoting methods, projected gradient descent applied to the NNLS subproblem does not have an obvious indication of convergence because numerical error will typically prevent its solution from satisfying the KKT optimality conditions exactly. To test for convergence we use the projected gradient threshold suggested by Lin [60]. Namely, if the inequality

$$\|\mathcal{P}_{NN}[\nabla f(W^{(k)}, H^{(k)})]\|_F \leq \epsilon \|\mathcal{P}_{NN}[\nabla f(W^{(0)}, H^{(0)})]\|_F \tag{2.19}$$

is satisfied at any point during algorithm execution, we assume the algorithm has converged and terminate it. In our implementation we set $\epsilon = 10^{-6}$, but this parameter may be varied to attain different algorithm behavior [60]. To account for cases when the algorithm takes a large number of iterations to converge, we impose a maximum of $50r$ on the number of iterations, although in the course of our extensive testing this maximum was almost never reached.

Like the active set and block principal pivoting algorithms, the projected gradient descent algorithm solves the NNLS problem by updating matrix columns in parallel for efficiency. However, it still may incur a large computational cost. The largest cost is due to finding a step size that satisfies Equation 2.17 during each iteration, which we assume requires an average of t tests of step sizes per iteration. To test each step size on the k-th iteration, we must first compute $\tilde{H}^{(k)} = \mathcal{P}_{NN}[H^{(k-1)} - \alpha_k \nabla f(H^{(k-1)})]$, which requires $O(pn)$ operations per test because the only entity that changes per test is α_k, so the other elements can be computed once and stored. However, computing $f(\tilde{H}) = \|X - W\tilde{H}\|_F^2$ requires $O(prn)$ operations per test, so the total cost per iteration of $O(tprn)$ operations is substantial.

To reduce this cost, Lin proposes a trick that takes advantage of Taylor's formula. Since $f(H)$ is quadratic, it is equivalent to its second-degree Taylor expansion. By rewriting $f(\tilde{H}^{(k)})$ as the Taylor expansion of $f(H^{(k-1)} + (\tilde{H}^{(k)} - H^{(k-1)}))$ [60], Equation 2.17 simplifies to

$$(1-\sigma)\langle \nabla f(H^{(k-1)}), \tilde{H}^{(k)} - H^{(k-1)} \rangle + \frac{1}{2}\langle \tilde{H}^{(k)} - H^{(k-1)}, (W^T W)(\tilde{H}^{(k)} - H^{(k-1)}) \rangle \leq 0 \tag{2.20}$$

The most significant cost to compute the LHS of the above equation is due to computing the matrix product $(W^T W)(\tilde{H}^{(k)} - H^{(k-1)})$, which is $O(r^2(p+n))$ operations. So the cost of using projected gradient descent to solve an NNLS subproblem in H is now an efficient $O(tr^2(p+n))$ operations times the number of iterations, plus the one-time $O(prn)$ cost of computing the matrices $W^T W$ and $W^T X$ necessary to compute the gradient, making for a total cost of

$$O(prn) + \text{number of iterations} \times O(tr^2(p+n)) \tag{2.21}$$

A similar argument about computing W to minimize $f(W) = \|X^T - H^T W^T\|_F^2$ yields the same cost result. Recall that this is only the number of operations required to solve the NNLS subproblem - we must multiply this number by the total number of NNLS subproblems we solve for convergence within the NNLS framework to attain the total number of operations required to compute a solution for NMF.

Algorithm 4: Projected Gradient Descent [60]

```
Input: Parameters $0 < \beta < 1$ and $0 < \sigma < 1$, function $f$, projection $\mathcal{P}$
Output: $A$ which is a local minimum of $f(A)$
  Initialization:
  Initialize any feasible $A$, set $\alpha_0 \leftarrow 1$
  for k = 1,2,... do
      if Stopping condition satisfied then
          Break
      end if
      Assign $\alpha_k \leftarrow \alpha_{k-1}$
      Compute $\tilde{A}^{(k)}$ by Equation 2.18
      if $\alpha_k$ satisfies Equation 2.17 then
          Set $\alpha_k \leftarrow \alpha_k / \beta$
          while $\alpha_k$ satisfies Equation 2.17 and $A^{(k)}(\alpha_k/\beta) \neq A^{(k)}(\alpha_k)$ do
              Set $\alpha_k \leftarrow \alpha_k / \beta$
          end while
      else
          Set $\alpha_k \leftarrow \alpha_k \cdot \beta$
          while $\alpha_k$ does not satisfy Equation 2.17 do
              Set $\alpha_k \leftarrow \alpha_k \cdot \beta$
          end while
      end if
      Set $A^{(k)} \leftarrow \mathcal{P}[A^{(k-1)} - \alpha_k \nabla f(A^{(k-1)})]$
  end for
```

2.4 Projected Gradient Descent (to directly solve NMF)

Lin's algorithm to apply projected gradient descent to solve NMF directly is the same algorithm, Algorithm 4, used to solve the NNLS subproblem, but with different parameters. For the direct application, $A \equiv (W, H)$, $f(A) \equiv f(W, H) \equiv \|X - WH\|_F^2$, and $\nabla f(A) \equiv (\nabla_W f(W, H), \nabla_H f(W, H))$.

By the same proofs of the convergence of the projected gradient descent method for the NNLS subproblem, the application of projected gradient descent here also converges to a stationary point. Nevertheless, we are well aware that a stationary point may not be the global minimum because of the nonconvexity of $f(W, H)$. For example, $(W, H) = (0, 0)$ is a stationary point often reached by the algorithm, although it is generally not a global minimum [60]. To avoid convergence to this point, initialization methods take an initial $W^{(0)}$ and solve $H^{(0)} = \arg\min_{H \geq 0} \|X - W^{(0)}H\|_F^2$ using any of the NNLS methods described in the previous section. This entails that

$$\|X - W^{(0)}H^{(0)}\|_F^2 \leq \|X - W^{(0)}0\|_F^2 = \|X\|_F^2 \tag{2.22}$$

Since the inequality is generally strict, this initialization will lead to a sequence that converges to a point with a smaller function value than $(0, 0)$, as desired [60]. However, the algorithm may still converge to a problematic stationary point; we test its tendency to do so in the following chapter.

Unlike projected gradient descent for NNLS, the function f is not quadratic, so the Taylor expansion trick cannot be applied. Consequently, computing $f(\tilde{W}, \tilde{H})$ requires $O(tprn)$ operations, which is the dominant cost per iteration.

Since this direct projected gradient descent algorithm updates W and H simultaneously, it is not a two-block coordinate descent algorithm. As we will see in Chapter 3, it tends to perform worse than all the two-block coordinate descent algorithms we test, unless it is initialized effectively, as we will see in Chapter 6.

2.5 Hierarchical Alternating Least Squares

Hierarchical alternating least squares (HALS) updates each column of W (and each row of H) using an exact coordinate descent method on each iteration [37]. It was originally developed by Cichocki et al in [24], but it has been rediscovered several times by multiple authors [37]. It first rewrites the NMF objective function as

$$\min_{W \geq 0, H \geq 0} \|X - \sum_{k=1}^{r} W(:, k)H(k, :)\|_F^2 \tag{2.23}$$

Next, HALS solves the NNLS subproblem over each column of W while holding the other columns constant to update W, and does the same over the rows of H to update H [24]. Each of these NNLS subproblems has a closed-form solution, since as we saw in our discussion of ANLS, the columns of H do not interact, so the NNLS subproblem for updating a row of H can be decoupled into n independent quadratic problems in one variable. A similar argument applies for updating W. In particular, for $l = 1, 2, ..., r$, the closed-form solution to update the l-th column of W is:

$$W(:,l) \leftarrow \text{argmin}_{W(:,l)\geq 0} \|X - \sum_{k\neq l} W(:,k)H(k,:) - W(:,l)H(l,:)\|_F \tag{2.24}$$

$$\leftarrow \max\Big(0, \frac{XH(l,:)^T - \sum_{k\neq l} W(:,k)(H(k,:)H(l,:)^T)}{\|H(l,:)\|_2^2}\Big) \tag{2.25}$$

Symmetrically, the closed-form solution to update the l-th row of H is:

$$H(l,:) \leftarrow \max\Big(0, \frac{X^T W(:,l) - \sum_{k\neq l} H(k,:)(W(:,k)^T W(:,l))}{\|W(:,l)\|_2^2}\Big) \tag{2.26}$$

$$\tag{2.27}$$

Under mild conditions, HALS is guaranteed to converge to a stationary point [84], and it tends to do so faster than most NMF algorithms [38]. The algorithm has a computational cost per iteration similar to the other NMF algorithms, namely $O(rpn+r^2(p+n))$ per iteration, due to the matrix multiplications and sums necessary to update each of $2r$ columns and rows between W and H.

Chapter 3

Tests of Older Heuristics

In this section we test how each of the NMF algorithms perform in relation to some particular quality of the data. Unless otherwise noted, we set $p = 50, r = 10$, and $n = 250$. For each test we initialize the initial estimates of W and H by sampling randomly from the absolute value of the normal distribution with mean 0 and standard deviation 1:

$$W_{ij} = |N(0,1)| \quad \forall i,j \tag{3.1}$$

$$H_{kl} = |N(0,1)| \quad \forall k,l \tag{3.2}$$

We rescale our initial estimate of W by the factor α^* introduced in Section 2.2 for consistency across all algorithms.

For stopping conditions, we implement the gradient thresholding technique described by Lin [60] to test for convergence in the NNLS subproblem solved by ANLS PGD, and test for whether all elements are optimal in the other two methods designed to solve the NNLS subproblem. We also set a maximum of 50 iterations allowed for each of these NNLS techniques. We do not implement sophisticated stopping conditions for any of the NMF algorithms; rather, we only impose a maximum on the number of iterations allowed. Doing so lets us to study error curves of the algorithms that extend as far into time as we desire for evaluation purposes.

Each plot is the average results over 40 trials. We refer to the algorithms that use the active set, block principal pivoting, and projected gradient descent methods within the ANLS framework as ANLS AS, ANLS BPP, and ANLS PGD, respectively, and projected gradient descent as PGD. To evaluate the algorithms, we track the relative error

$$\frac{\|X - WH\|_F}{\|X\|_F} \tag{3.3}$$

where W and H are the factor estimates repetitively updated by the algorithms throughout their execution. All experiments are run on a 2.7 GHz Intel Core i5 processor with 8 GB of memory via 1867 MHz DDR3. We start by running simple tests of the algorithms on randomly generated Gaussian data in order to familiarize the reader with and provide a baseline for algorithm performance.

3.1 Vanilla Gaussian Data

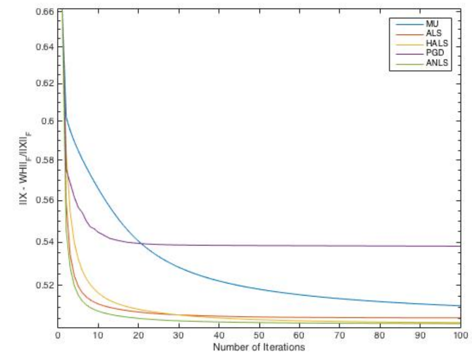
For the first tests we generate X from the same distribution that we generate our initial estimates of W and H, namely

$$X_{ij} = |N(0,1)| \quad \forall i,j \tag{3.4}$$

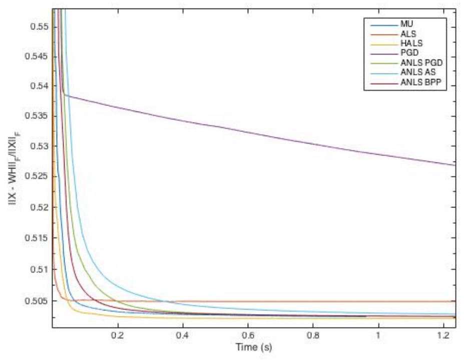
Here X has $p = 50$ rows and $n = 250$ columns, and compute a rank-10 NMF of X, such that W has $r = 10$ columns and H has $r = 10$ rows. Figure 3.1(a) shows the relative error of five NMF algorithms after each iteration over 100 iterations. We initialize each algorithm with the same W and H, so on every iteration each of the three ANLS algorithms finds the same unique solution to the two NNLS subproblems[1]. Thus, their relative error on every iteration is by default the same, hence we use the same curve to denote all of them. Figure 3.1(b) shows the relative error versus time for each algorithm for a maximum of 8000 iterations. Since each of the three ANLS algorithms solve the NNLS subproblems with varying efficiency, their convergence differs versus time, which we show in three separate curves. Finally, Figure 3.2 shows the cumulative time that each algorithm takes to run over 8000 iterations for the non-ANLS algorithms, and 500 iterations for the ANLS algorithms.

Over the first 100 iterations ANLS, HALS and ALS all perform similarly well versus the number of iterations, while MU and PGD converge much slower per iteration. However, the algorithms' convergence versus time is more revealing of their performance. Each iteration of ALS executes very quickly, but by about 30 iterations ALS reaches a limit point and stops converging. Over time, this allows HALS, MU, and each of the ANLS algorithms to converge to a smaller error than it. PGD converges so slowly (much slower than all of the other algorithms) that its ultimate error upon reaching a limit point is practically irrelevant. For reference, though, in the tests we ran, it does not surpass the other algorithms in terms of relative error, but instead

[1]Unless for some reason they do not solve the NNLS subproblem before reaching the maximum number of iterations, as observed in Section 3.3.



(a)



(b)

Figure 3.1: Relative error vs (a) number of iterations and (b) time

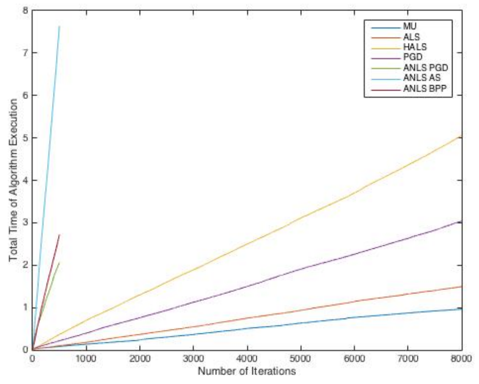


Figure 3.2: Cumulative time vs number of iterations.

plateaus before reaching the other error curves. Meanwhile, the low computational cost of each MU iteration compensates for its small improvement per iteration, allowing it to outperform every other algorithm besides HALS versus time. Indeed, HALS performs the best versus time. Among the ANLS algorithms, the block-principal pivoting method executes the fastest per iteration with PGD a close second.

In general, these results support most of our hypotheses about algorithm performance discussed in Chapter 2. We expect HALS and ANLS to converge relatively quickly per iteration since they solve the NNLS subproblems exactly on each iteration, and ALS, MU and PGD to converge significantly less per iteration. We also expect ALS and MU to take the least time to run per iteration since they have the least number of operations per iteration, and the ANLS algorithms to take the most time for the opposite reason. Finally, among the ANLS algorithms we expect that the active set method has the largest computational cost because it only transfers at most one variable between active and passive sets per column per iteration. On the other hand, the fast convergence of ALS over its first 100 iterations is surprising, but what makes more sense is the fact that ALS stops converging at a more erroneous solution than the other algorithms, since the projected least squares method is not provably optimal. The fast convergence of MU over time is also rather unexpected. Of course, the performance of each of these algorithms as observed here may be particular to the Gaussian dataset with this dimension, so to gain a more complete understanding we test algorithm performance over a variety of other types of data next.

3.2 Dimension

Here we vary the number of rows p and columns n of X while keeping every other parameter the same as the previous section and test algorithm performance. We first vary the number of rows while keeping the number of columns constant at $n = 250$. In Figure 3.3, we plot algorithm relative error versus time for three separate numbers of rows in the data matrix, and in Figure 3.4 we plot the relative error yielded by ANLS BPP versus time for each of the three possible numbers of rows on the same graph.

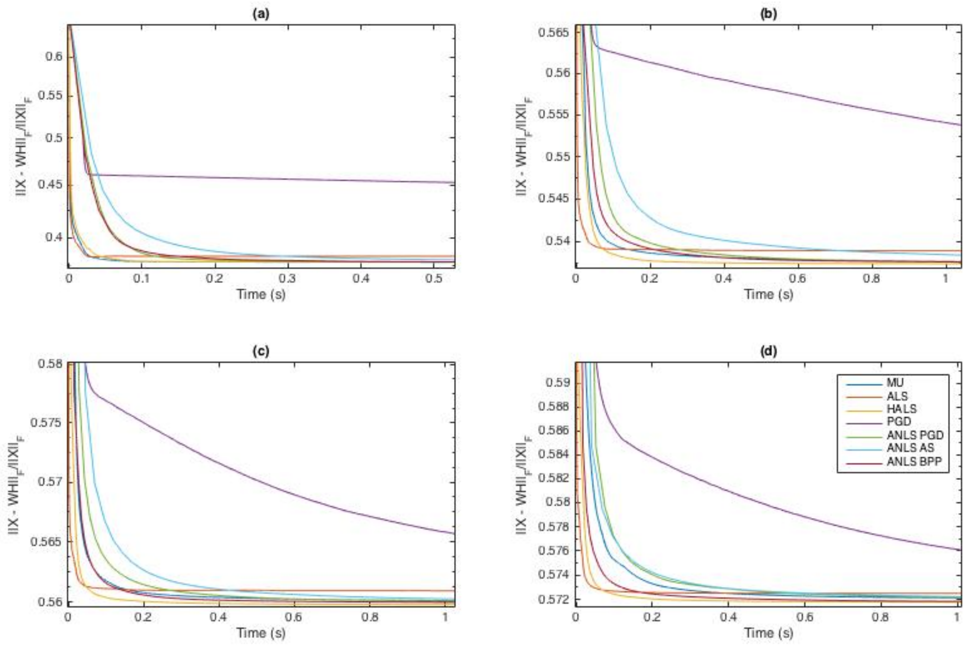


Figure 3.3: Algorithm performance with (a) $p = 20$, (b) $p = 100$, (c) $p = 250$, (d) $p = 500$.

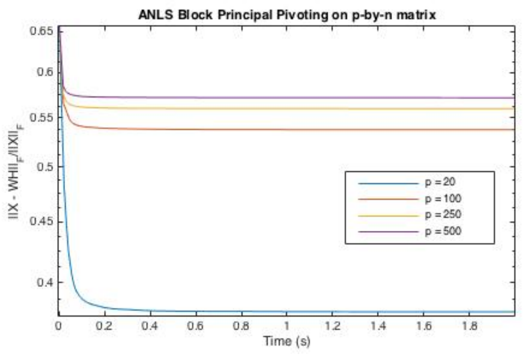


Figure 3.4: Performance of ANLS BPP vs time (s) with varying number of rows in the data matrix.

There are multiple key takeaways from these plots. First, HALS performs marginally the best across all tests, followed closely by MU and ALS, and each of the algorithms perform worse as the number of rows increases. PGD performs significantly worse than any other algorithm, but scales the best with the number of rows; surprisingly, it converges faster to its final relative error as the number of rows increase. It is still slower than every other algorithm even at the largest number of rows, but the gap has substantially decreased. The scalability of section PGD does not apply for its ANLS version, however, as it performs comparatively worse as the number of rows increases. Likewise, the number of algorithms which outperform MU also increase with the number of rows. Lastly, the larger the number of rows, the less the increase in rows appears to cause the final relative error of the algorithms to increase, which is observable especially in Figure 3.4.

The results when we vary the number of columns n and hold the number of rows p constant at $p = 250$ are very similar to the results when varying the number of rows. The only significant difference is that the relative performance of PGD diminishes (instead of improves) as the number of columns increases, as we see in Figure 3.5. This surprising phenomenon is likely not a result of random variation in algorithm performance due to the random generation of the the data and initialization, since the results are averaged over 40 trials.

3.3 Nonnegative Rank

We next vary the nonnegative rank of the data matrix X and evaluate algorithm performance. Let k denote the nonnegative rank of X and r denote the factorization rank, namely $\text{rank}(WH)$. We generate a nonnegative rank-k matrix $X \in \mathbb{R}^{p \times n}$ by

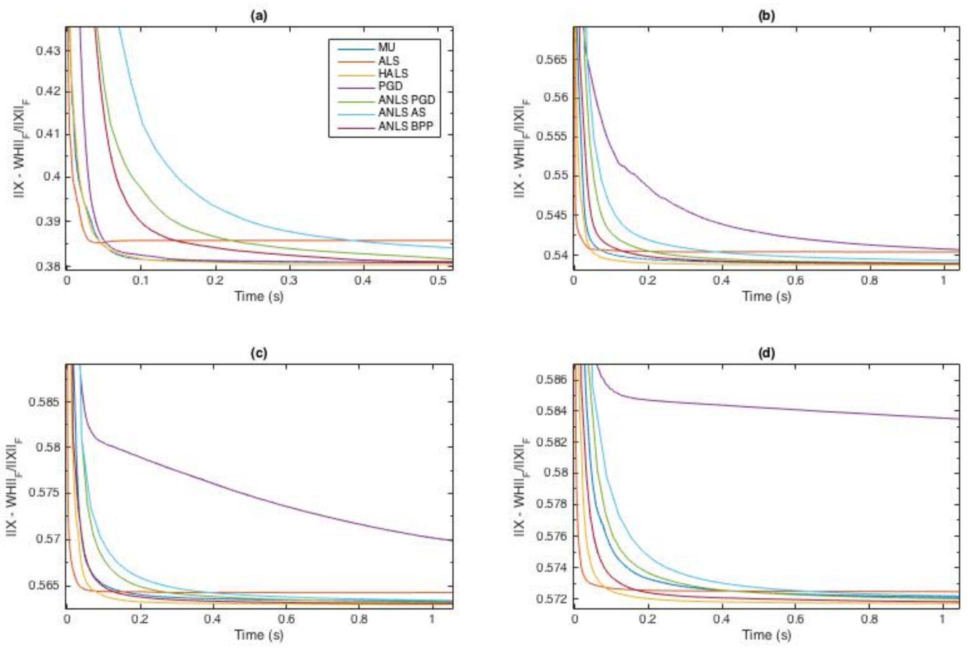


Figure 3.5: Algorithm performance with (a) $n = 20$, (b) $n = 100$, (c) $n = 250$, (d) $n = 500$.

randomly generating matrices $U \in \mathbb{R}_{\geq 0}^{p \times k}$ and $V \in \mathbb{R}_{\geq 0}^{k \times n}$ where $u_{ij}, v_{st} = |N(0, 1)|$ for all (i, j) and (s, t), then computing their product $X = UV$. Again, $p = 50$ and $n = 250$, and the initialization procedure is the same as in previous tests.

We hold the factorization rank constant at $r = 10$ for tests over the data rank k. The results are displayed in Figure 3.6. We place a maximum on the number of iterations allowed of 30,000 for MU, 18,000 for ALS, 12,000 for PGD and HALS, and 1,000 for the ANLS algorithms, explaining why some of the curves terminate within the figure.

When $k < r$, most NMF algorithms find an NMF that very closely approximates X. However, the speed and accuracy with which they do this, even among those algorithms that find close approximations, varies greatly. HALS converges the fastest to an exact solution in both of these instances, whereas ALS finds a close but definitely not optimal solution and oscillates around it.

Meanwhile, the behavior of the ANLS BPP and ANLS AS methods is intriguing. If either ANLS AS or ANLS BPP reaches the maximum number of subiterations to solve the ANLS subproblem, this implies that the algorithm did not solve the NNLS subproblems exactly since the number of not optimal elements was never zero on any subiteration. Both ANLS AS and ANLS BPP reached the maximum number of

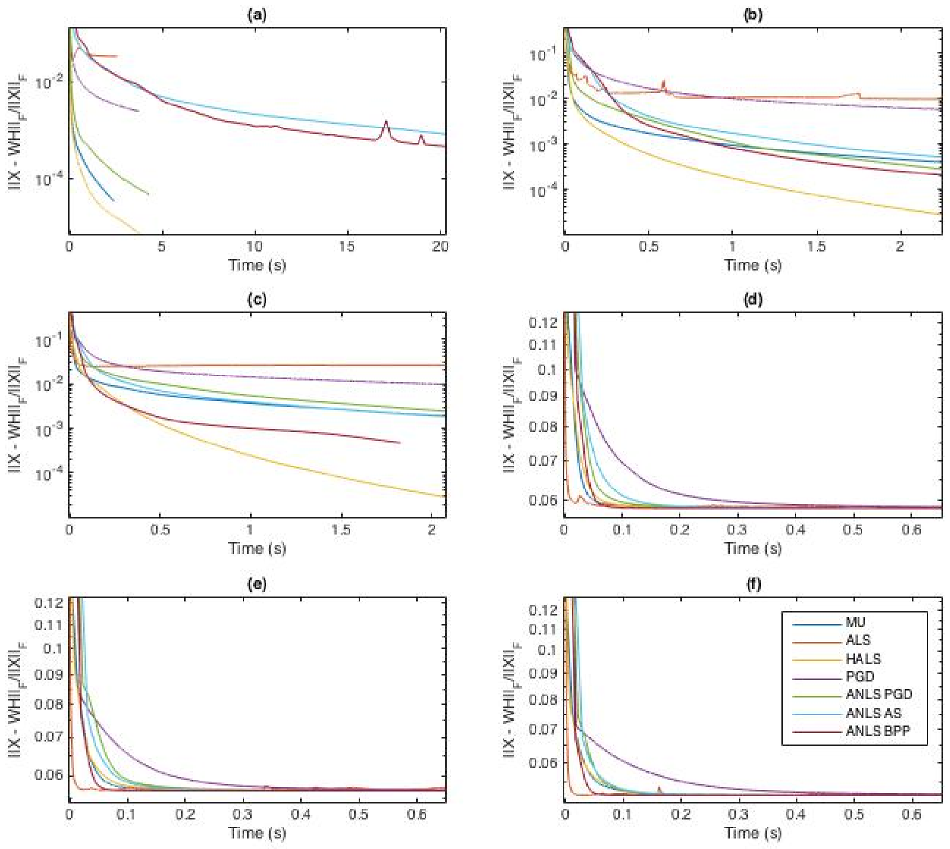


Figure 3.6: Algorithm performance with (a) $k = 4$, (b) $k = 7$, (c) $k = 10$, (d) $k = 24$, (e) $k = 37$, (f) $k = 50$.

subiterations (50) on an average of 71.4 and 46.0 % of attempts to solve the NNLS subproblems in H when $k = 4$, respectively, and 1.89 and 0.24 % when $k = 7$. Moreover, ANLS AS reached the maximum number of subiterations an average of 2.64 % of iterations when solving for W when $k = 4$ (in all other instances ANLS AS and ANLS BPP never reached the maximum number of subiterations). Thus, neither ANLS BPP nor ANLS AS solves the NNLS subproblems exactly on every iteration when $k < r$.

Why might the algorithms not solve the NNLS subproblems exactly in less than the maximum number of iterations? We claim it is due to ill-conditioning of the factor matrices. First consider the NNLS subproblem to solve for H, which involved many more not optimal solutions than the subproblem in W. During the execution of both algorithms, rank(W) approaches $k < r$ in order to more accurately represent the features of X. This means W becomes singular, thus the matrix $W^TW \in \mathbb{R}^{r\times r}$ is very ill-conditioned. In turn, this entails that the normal equations $W^TWH = W^TX$ in the ANLS AS and ANLS BPP algorithms cannot be solved accurately. This leads to numerical error, so the algorithms do not necessarily find an optimal solution. Another way to interpret this is that the algorithms are trying to overfit parameters (elements in W and H) to an underdetermined system. Doing so means that there is no clear optimal way to assign the extraneous parameters, leading to the algorithms not being able to find an optimal solution.

The algorithms likely rarely reached the maximum number of subiterations when solving NNLS subproblems in W because H does not necessarily tend to mimic the rank of X, and it is harder for $\text{rank}(H) < r$ because the number of columns of H is generally much larger than the number of rows of W. So the matrix $HH^T \in \mathbb{R}^{r\times r}$ is typically non-singular and well-conditioned, thus the error due to ill-conditioning during algorithm execution is usually trivial.

The algorithms still run and may converge to an optimal solution, but their error due to the ill-conditioning of W^TW means that they may do so somewhat erratically. The relative error yielded by ANLS BPP spikes for both $k = 4$ and $k = 7$ (the latter spike is out of the figure), and the relative errors yielded by ANLS AS eventually jumps to around 10^{-1} (after a long time, > 150 seconds of algorithm execution for $k = 4$ and > 10 seconds for $k = 7$, which are out of the figures). The relative error yielded by ANLS BPP also jumps for $k = 4$ (again this is out of the figure). We remain unsure as to what exactly causes this spiking and jumping behavior, but we suspect it has something to do with the ill-conditioning of the factor matrices.

Another interesting note is that ANLS AS and ANLS BPP take drastically longer to execute when $k = 4$ than otherwise. They take approximately 321 and 170 seconds, respectively, to execute 1000 iterations when $k = 4$, and 13 and 7.5 seconds, and 9 and 2 seconds, to execute the same number of iterations when $k = 7$ and 10, respectively. This is likely also a product of the algorithms reaching the maximum number of subiterations on a much larger%age of NNLS subproblems when $k = 4$ than otherwise.

Finally, we would like to call attention to the fact that there is not much difference in algorithm performance between all tests with $k > r$, especially for HALS, PGD, and ALS. Likewise, the performance of each of the algorithms changes very little between tests in which $k > r$. In particular, each of the algorithms converge relatively quickly to relative errors of 0.0583, 0.0565, and 0.0519 when $k = 24, 37$, and 50, respectively. This suggests that the NMF problem with factorization rank r for a data matrix X that has a nonnegative factorization of rank k has a phase shift at $k = r$; as long as $k \leq r$, the NMF algorithms tend to find close to exact solutions regardless of the choice of $k \leq r$, yet as soon as $k > r$, each of the algorithms converges to a similar relative error for any $k > r$. In turn, the lack of significant decrease in performance as k increases further past r implies that X has $\leq r$ important principal components for each value of k that we tested, so a rank-r nonnegative factorization of it can represent it fairly well.

3.4 Condition Number

We next test over the condition number of the data X, where we define the condition number $\kappa(X)$ with respect to the spectral norm:

$$\kappa(X) = \|X\|\|X^{-1}\| = \frac{\sigma_{max}(X)}{\sigma_{min}(X)} \tag{3.5}$$

To generate a random matrix $X \in \mathbb{R}^{p\times n}$ with a particular condition number α, we first generate a matrix $Y \in \mathbb{R}^{p\times n}$ in which each element is sampled from $|N(0,1)|$. Next we scale the singular values of Y. Let $UDV^T = Y$ be the full singular value decomposition of Y. Note that the diagonal elements of D (the singular values of Y) are ordered such that d_{11} is the largest and d_{nn} is the largest For a given condition number α, we compute the updated diagonal matrix $D^{(\alpha)}$ such that

$$d_{ii}^{(\alpha)} = d_{11}(1 - \frac{\alpha - 1}{\alpha}\frac{d_{11} - d_{ii}}{d_{11} - d_{nn}}) \tag{3.6}$$

for all $1 \leq i \leq n$. Then the matrix $X = UD^{\alpha}V^T$ has condition number α. To see this, first note that by construction of Y, none of its singular values are zero nor equivalent to each other. Next, since the RHS of 3.6 is monotone decreasing in i, and the diagonal elements of $D^{(\alpha)}$ are the singular values of $X = UD^{(\alpha)}V^T$, $d_{11}^{(\alpha)}$ is the largest singular value of X and $d_{nn}^{(\alpha)}$ is the smallest. Thus we have:

$$\kappa(X) = \frac{d_{11}^{(\alpha)}}{d_{nn}^{(\alpha)}} = \frac{d_{11}}{\frac{d_{11}}{\alpha}} = \alpha \tag{3.7}$$

Furthermore, as long as $\alpha > \kappa(Y)$, X will be nonnegative. This follows first from recalling that by the definition of the SVD, X is the weighted sum of its singular factors: $X = \sum_{i=1} d^{(\alpha)} u_i v_i^T$. Since Y is nonnegative, the singular factor $u_1 v_1^T$ is also nonnegative by Perron-Frobenius theory [69]. Note that Equation 3.6 does not change the weight (d_{11}) of this singular factor in X. It only scales the other singular values, whose corresponding singular factors may be negative. If it downscales these singular values, which happens when $\alpha > \kappa(Y)$, then the negative elements in the factors cannot outweigh the positive ones in $u_1 v_1^T$ and other factors. However, if it upscales the other singular values, as it does when $\alpha < \kappa(Y)$, then negative elements in some of the factors may gain enough weight to outweigh the positive elements.

We generate the matrix X as described, with $p = 50$ and $n = 250$, and test how well each of the seven NMF algorithms factor $X^{(\alpha)}$ for values of alpha in the set $\{20, 50, 100, 10,000\}$. Each of these values of α is larger than the condition number of the initially-generated X (by taking the absolute value of Gaussian samples). The results are shown in Figure 3.7.

Each of the plots have nearly the same curves, but each curve is shifted downwards by the same relative error as the condition number of X increases. This uniformity implies that the condition number of X does not change the descent pattern of the NMF algorithms that factor it, but it does uniformly affect the relative error yielded at all iterations of each algorithm. It makes sense that the algorithms perform better with larger condition numbers, since a larger condition number implies the matrix is likely closer to being low rank, so the low rank representation of the matrix given by the NMF is likely to be more accurate than if the matrix had a smaller condition number and larger rank. It also makes sense that downward shift size decreases dramatically as the condition number of X increases, based on our previous results from factoring X with varying ranks.

3.5 Heavy-Tailed Data

A heavy-tailed distribution is a probability distribution with at least one tail that is not exponentially bounded, meaning they have a higher probability of generating outliers than the exponential distribution. Such distributions accurately model many real phenomena, including financial loss, medical costs, birth weights, and rainfall, just to name a few [73].

Here we test algorithm performance on four datasets composed of elements independently drawn from distinct heavy-tailed distributions. The four distributions that we test are the (a) log-normal, (b) Weibull, (c) Pareto, and (d) log-logistic distributions. After choosing parameters to specify the most fundamental versions of these

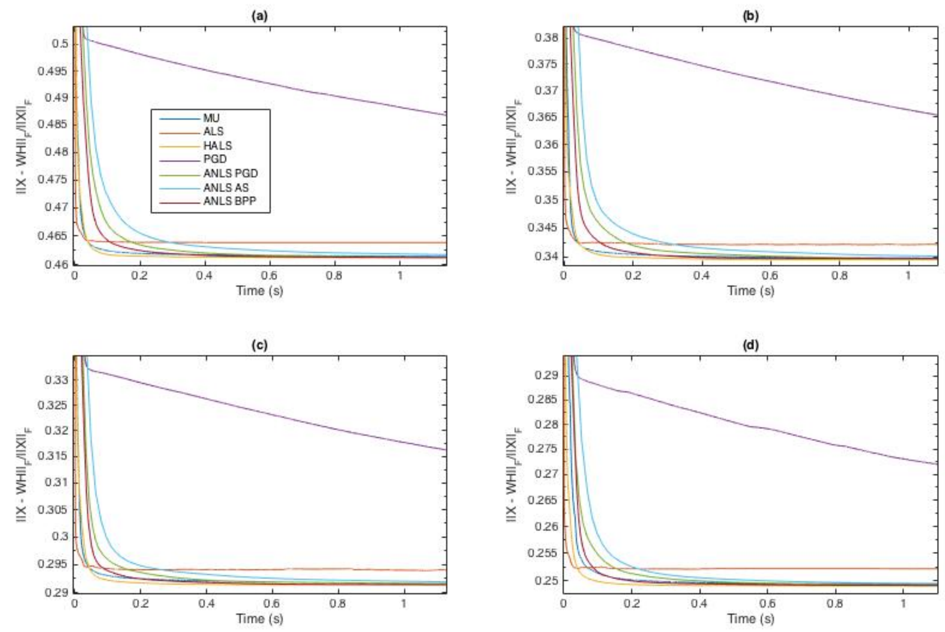


Figure 3.7: Algorithm performance on data with varying condition number α, such that in (a) $\alpha = 20$, (b) $\alpha = 50$, (c) $\alpha = 100$, (d) $\alpha = 10,000$.

distributions, their respective density functions are

$$\begin{aligned}
&\text{a)} \quad p(x) = \frac{1}{x\sqrt{2\pi}} e^{-(\ln x)^2/2} \text{ for } x \geq 0 \\
&\text{b)} \quad p(x) = \frac{1}{2} x^{-1/2} e^{-x^{1/2}}, \text{ for } x \geq 0 \\
&\text{c)} \quad p(x) = \frac{1}{x^2}, \text{ for } x \geq 1 \\
&\text{d)} \quad p(x) = \frac{1}{(1+x)^2}, \text{ for } x \geq 0
\end{aligned} \tag{3.8}$$

Importantly, all of these distributions exist only over nonnegative values, so they fit the nonnegativity assumption on X. In part because they are nonnegative, the distributions are also one-tailed. We generate independent identically-distributed samples from these distributions to populate the X matrix using MATLAB's `random` function.

Notably, the distribution given by the absolute value of the normal distribution has tails lighter than those of the exponential distribution, so our tests in Section 3.1 evaluated algorithm performance on light-tailed data. To enable direct comparison with our results from Section 3.1, we hold all other parameters the same from the tests in that section. Namely, we maintain the problem dimensions of $p = 50$, $n = 250$, and $r = 10$, and generate our initial estimates of W and H according to the same procedure of taking the absolute value of independent Gaussian samples.

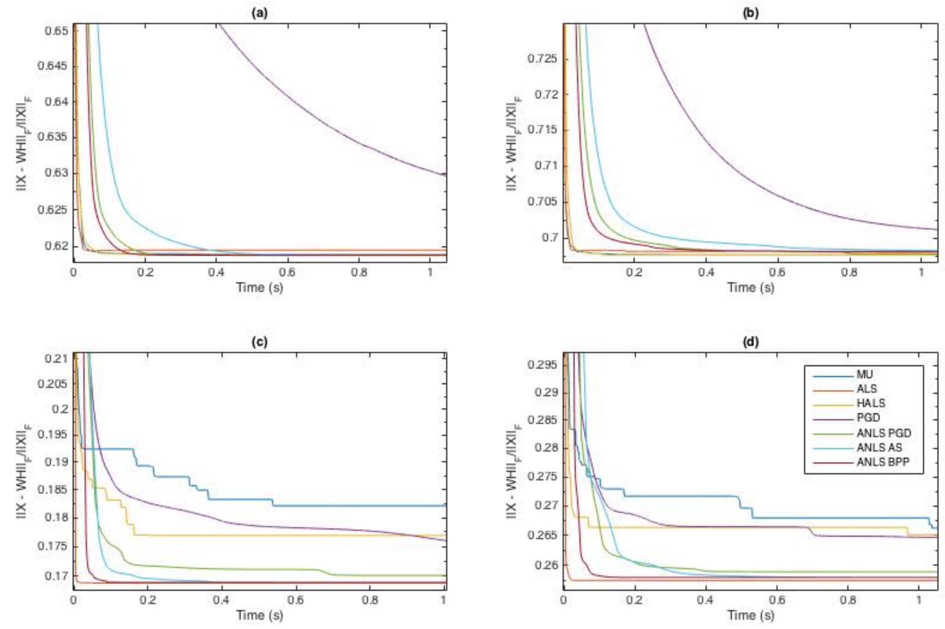


Figure 3.8: Algorithm performance in terms of relative error vs time for data drawn from (a) log-normal, (b) Weibull, (c) Pareto, and (d) log-logistic distributions.

Figure 3.8 shows the relative error over time yielded by the seven NMF algorithms for each of the four heavy-tailed distributions. The results for the log-normal and Weibull data are very similar to the results from the light-tailed (normal) data, with the only significant difference being that MU and ALS slightly outperform all the other algorithms, including HALS and ANLS BPP, for the Weibull data.

In contrast, the performance of the algorithms on the Pareto and log-logistic data follows a novel structure. First, HALS, MU, and to a lesser extent, both PGD methods, descend in choppy steps (even after averaging the descent curves over 10 trials as usual), unlike the smooth descent patterns seen previously. This may indicate that NMF of Pareto and log-logistic data has a rougher optimization landscape than that of previously tested distributions. Secondly, MU and HALS perform much worse compared to the other algorithms than in previous steps, suggesting that algorithm performance and choppiness of descent may be somehow related. Lastly, sectionPGD and ALS perform noticeably better than usual for these two distributions - in fact, ALS appears to be the optimal algorithm among the seven to solve NMF for Pareto and log-logistic data.

3.6 Sparsity

In this section we evaluate algorithm performance on data that has varying levels of sparsity. We calculate sparsity rates as the fraction of elements in X that are zero. A variety of NMF techniques exist to enforce sparsity on the factors W and H [37], but here we only care about how the sparsity of X affects fundamental algorithm performance when W and H are not subject to any sparsity constraints.

To generate $X \in \mathbb{R}^{p\times n}_{\geq 0}$ with sparsity rate s, we randomly choose a set of $pn(1-s)$ indices of X, then set the element at each of these indices to be equal to the absolute value of a random sample from the normal distribution. We set the other elements equal to 0. Again, we use $p = 50$, $n = 250$, and $r = 10$. We share our results in Figure 3.9.

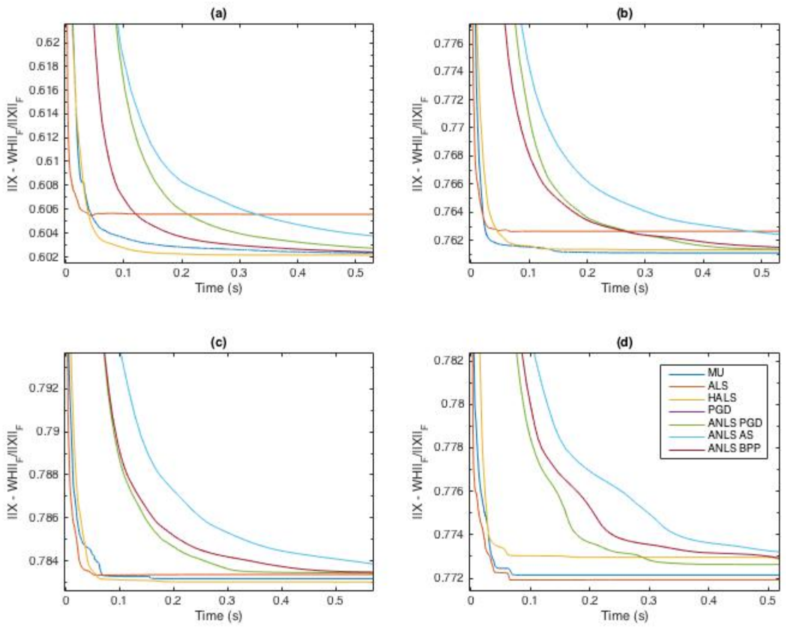


Figure 3.9: Algorithm performance on data with sparsity rate equal to (a) 0.25, (b) 0.75, (c) 0.9, and (d) 0.95.

For sparsity rates of 0.25, 0.75 and 0.9, the relative error curves follow a familiar pattern, with HALS and MU performing the best and PGD the worst (its relative error is too large to fit in the displayed plots). For a sparsity rate of 0.95, however, ALS slightly surpasses the previously top two algorithms. On a related note, although HALS improves its performance for a sparsity rate of 0.95 from its performance for a sparsity rate of 0.9, its relative performance compared to all of the other algorithms diminishes substantially. Indeed, the relative errors for a sparsity rate of 0.95 decrease from those for a sparsity rate of 0.9, whereas they had previously increased with sparsity rate, suggesting that NMF algorithm performance decreases with more sparsity in the data until that sparsity level reaches a critical point, after which algorithm performance improves.

3.7 Missing Data

We again test NMF algorithm performance on data with varying numbers of zeros, but here the zeros are problematic: they represent missing elements from the data. Our goal is to evaluate how well NMF algorithms can solve the matrix completion problem of recovering the matrix X when they only observe a subset of its elements indexed by the set Ω. In other words, the observed matrix is $\mathcal{P}_\Omega(X)$, where

$$\mathcal{P}_\Omega(X)_{ij} = \begin{cases} x_{ij}, & \text{if } (i,j) \in \Omega \\ 0, & \text{otherwise} \end{cases}$$

and compute an NMF (W, H) of $\mathcal{P}_\Omega(X)$, and evaluate how well WH represents the true data matrix X for each algorithm and over varying missing entry rates. Algorithms designed specifically to solve the matrix completion problem would attempt to find the minimum over (W, H) of the function $\|\mathcal{P}_\Omega(X - WH)\|_F^2$, but here the NMF algorithms instead try to minimize $\|\mathcal{P}_\Omega(X) - WH\|_F^2$, since trying to optimize the former function would require substantially altering the NMF algorithms. Algorithm performance is evaluated based on the relative error of WH from the ground truth matrix X. We set $r = 10$, and construct X wiht 50 rows and 250 columns, and with elements equal to the absolute values of samples from the standard normal distribution. Given a cardinality, we compose Ω by sampling without replacement from a uniform distribution over the indices of X using the MATLAB function `datasample`. Noting that the missing entry rate equals $1 - \dfrac{|\Omega|}{pn}$, the results for four different missing entry rates are shown in Figure 3.10.

The results are intriguing in a number of ways. First, in a rare occurrence, ALS performs the best among the seven algorithms for all missing entry rates. This superior performance may have to do with ALS's projection onto the nonnegative orthant,

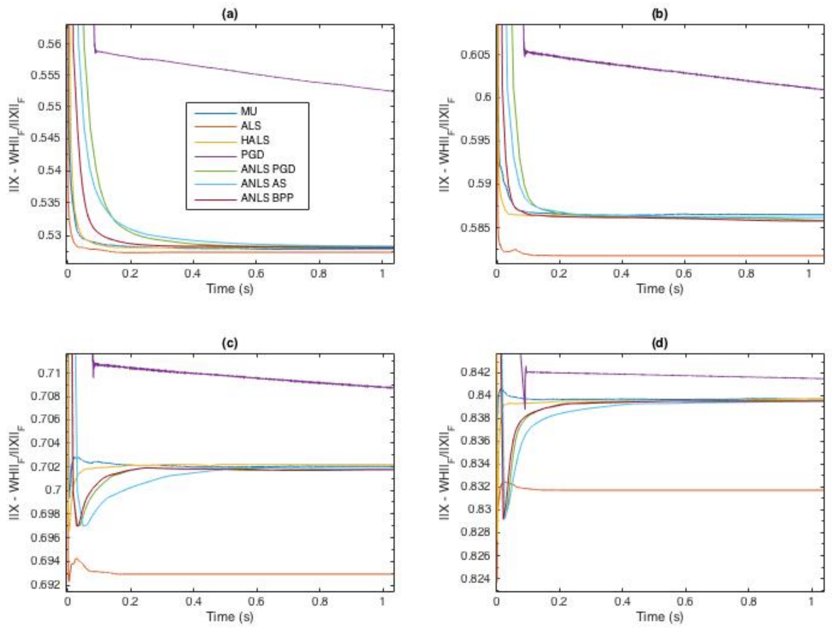


Figure 3.10: Algorithm performance on missing data with missing entry rates equal to (a) 0.1, (b) 0.25, (c) 0.5, and (d) 0.75.

which may allow greater flexibility in fitting the product of estimates WH more precisely to the non-missing entries while letting the estimates of missing entries be very negative. In fact, ALS and the rest of the algorithms do a fairly well at recovering X; their final relative errors are larger than the missing entry rates, but they are never more than 1, even when the missing entry rate is as high as 0.75. On this note, each algorithm's final ultimate relative error for each missing entry rate increases roughly linearly with the missing entry rate, and the error rates decrease substantially from the initial random estimates (this is the case even for a sparsity rate of 0.75, although it is difficult to tell in Figure 3.10(d)).

Moreover, the algorithms' relative error curves exhibit overfitting that becomes more pronounced as the missing entry rate increases, starting with the missing entry rate of 0.5. This overfitting means that at start of the algorithms' convergence towards $\mathcal{P}_\Omega(X)$, they also happen to converge to the ground truth matrix X, explaining the initial decrease in relative error from X, but after more iterations they pass X on their way to $\mathcal{P}_\Omega(X)$, explaining the sudden increase in relative error from X. Such behavior is interesting and presents a potential avenue for further investigation.

Finally, PGD oscillates a very small, seemingly patternless amount on each iteration, and the magnitude of the oscillation increases with missing entry rate. Perhaps this suggests that the descent direction towards $\mathcal{P}_\Omega(X)$ may also happen to be towards X on one iteration, then away from X on the next, and the variance in descending towards X increases as $\mathcal{P}_\Omega(X)$ and X become further apart.

3.8 Noisy Data

Next we test each algorithms' robustness to noise, which is a generalization of robustness to missing data. We generate the matrix $X \in \mathbb{R}^{p\times n}$ as usual and generate the noise matrix N_ζ where each element is independently sampled from the distribution $N(0,\zeta)$ for various values of ζ. Then we take the projection of their sum onto the nonnegative orthant. Namely, the matrix we input to the NMF algorithms is $\mathcal{P}_{NN}(X+N)$, where $\mathcal{P}_{NN}(A)$ is again defined as the element-wise maximum $\max(A,0)$. The power of a signal with mean 0 and variance ζ^2 is equal to ζ^2 (before taking the absolute value of the signal, and taking the absolute value does not change the power), so we can calculate the signal-to-noise ratio (SNR) in decibels before the projection as

$$\text{SNR} = 10\log_{10}\frac{P_X}{P_N} = 10\log_{10}\frac{1}{\zeta^2} \tag{3.9}$$

Again, we evaluate how our algorithms perform in terms of the relative error between X and WH, and plot the results for four different pre-projection SNRs in Figure 3.11.

The results are similar to those for missing data, with some notable differences. ALS again performs exceptionally well, although PGD outperforms it for nonnegative pre-projection SNRs, despite the fact that PGD diverges from the optimal solution. Like in the test with missing entries, PGD exhibits very small oscillations between iterations.

Overfitting again occurs, especially at the higher SNRs. At the lower SNRs, the algorithms tend to diverge from the ground truth solution right away, instead of first approaching it and then decreasing as in the overshooting case. Recall our prior hypothesis that the overshoot indicates that the algorithms initially happen to converge to ground truth matrix by virtue of the fact that it is in the same direction as the noisy matrix, then pass the ground truth matrix and move away from it in the course of their convergence to the noisy matrix. The results in Figure 3.11 support this hypothesis. The more distant the noisy and ground truth matrices are from each other, which we assume increases with decreasing SNR, the less likely the estimates yielded by the algorithms are to happen to approach the ground truth matrix on

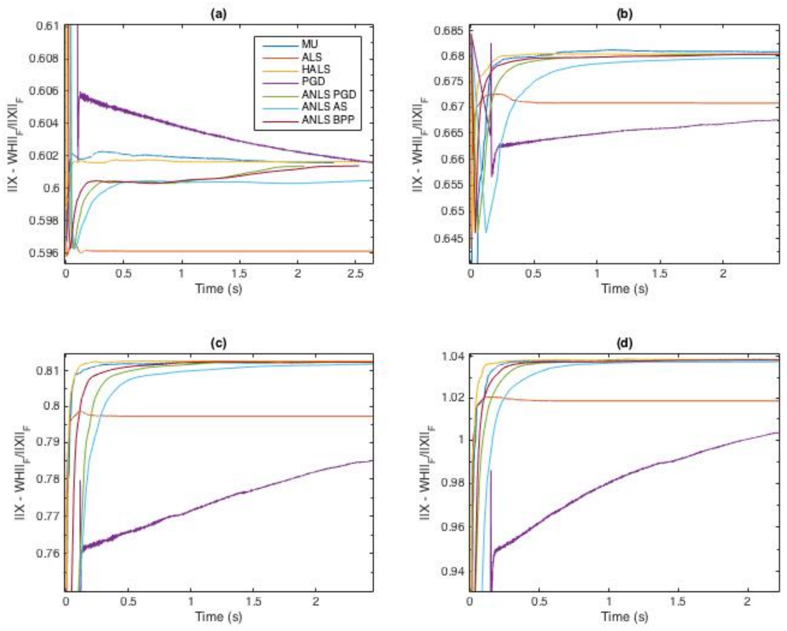


Figure 3.11: Algorithm performance on noisy data with pre-projection SNRs equal to (a) 3 dB, (b) 0 dB, (c) −3 dB, and (d) −6 dB.

their paths to the noisy matrix. This means that the overshoot behavior is less likely at lower SNRs; instead, the relative error curves directly increase without decreasing significantly, as in Figure 3.11.

Notably, for negative SNRs the algorithms increase in terms of relative error right away, and never substantially decrease. This means that the random initialization of (W, H) yields a better estimate of X than do the NMF algorithms executed on the noisy matrix, and suggests that if the SNR is known to be negative, NMF should not be employed.

3.9 Binary Data

Any categorical data that signifies membership in one of two classes is binary (all entries are 0 or 1), such as the educational data we will study in the next section. Consequently, binary data is pervasive, raising the question of how well NMF algorithms can factor binary data, which we will explore in this section.

We generate binary data such that each entry has a probability q of being 0 and is generated independently from every other entry. We show the factorization results in Figure 3.12. It is helpful to compare these results to the results from the tests

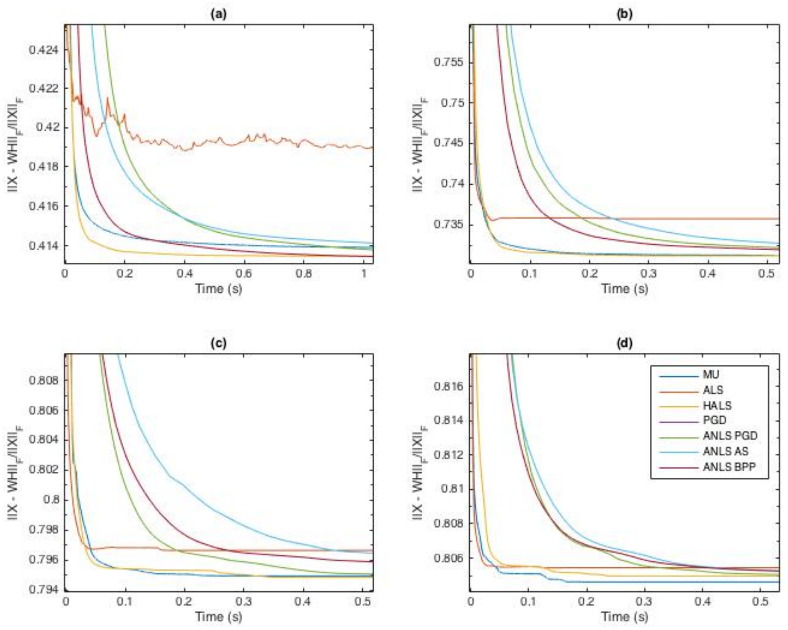


Figure 3.12: Algorithm performance on binary data with (a) $q = 0.25$, (b) $q = 0.75$, (c) $q = 0.9$, (d) $q = 0.95$.

over the sparsity of the data, because the only significant difference between the two sets of tests is that the nonzero entries can be any positive number in the tests over sparsity, but here they are restricted to be equal to 1 (the other difference is that here the number of zero entries is stochastic, but after averaging over 10 trials it is close enough to q to enable meaningful comparisons).

When the rate of zero entries is approximately 0.25, the final relative error rates achieved by the algorithms are much lower for binary data than they are for sparse data (0.414 vs. 0.602, resp.). Otherwise, the results are comparable. In particular, the error rates similarly increase with larger percentages of zeros for binary data. However, unlike the tests over sparse data, performance does not improve from rates of approximately 0.9 zeros to 0.95 zeros. Also of note is that ALS is again the only algorithm whose relative performance improves with higher rates of zeros in the data. One broad takeaway from all of our tests is that the more degenerate the data, the better ALS seems to perform compared to its competitor algorithms. Furthermore, HALS performs the best for small q, but is surpassed by MU for $q \in \{0.9, 0.95\}$. Finally, one may suggest that initializing W and H to be random binary matrices with probability of entries being zero equal to q would lead to a more accurate factorization, but we find that this initialization yields a significantly more erroneous factorization than our random Gaussian initialization in all cases.

3.10 Summary of Main Findings

The main takeaways from the experimental results discussed in this section are as follows:

- For vanilla Gaussian data, HALS performs marginally the best, and ANLS BPP is the fastest algorithm among the ANLS methods. ALS has fast initial convergence but reaches a poor local minimum, and PGD converges extremely slowly.
- The larger the dimensions of X, the more erroneous the NMF solutions yielded by all the algorithms, but the marginal increase in this error diminishes with larger dimensions.
- When the nonnegative rank of X is at most the factorization rank, the NMF algorithms find a solution that very closely approximates X, and HALS is the best at this. ANLS AS and ANLS BPP suffer from ill-conditioning when the nonnegative rank is less than the factorization rank. When the nonnegative rank is greater than the factorization rank, ALS converges the fastest, but all algorithms converge to the same relative error.
- Algorithm performance increases with larger condition numbers of X across all algorithms, with PGD and ALS performing the worst.
- Algorithm performance on datasets sampled from the log-normal and Weibull distributions is similar to the Gaussian data for all algorithms. Conversely, the algorithms perform much better on the Pareto and log-log distributions, especially ALS and ANLS BPP, while MU yields the largest relative errors for these datasets.
- ALS performs the best on very sparse data, followed closely by MU. All algorithms perform worse with increasing sparsity, but HALS significantly diminishes in performance relative to the other algorithms as the sparsity of the data increases.
- All algorithms exhibit overfitting on missing data, and ALS performs the best in this case by a large margin.

- In the case of more general noise, overfitting again occurs, and ALS again converges to the most stable solution.
- ALS performs the best when binary data is highly sparse; for less sparse binary data, HALS and MU are the most effective.

Chapter 4

Case Study: Educational Data

As mentioned in Section 1, there are many well-studied applications of NMF. One application area that theory suggests should be fruitful is using NMF to recover latent variables from educational datasets. The most prevalent form of educational data is students' test question results, captured in a question-by-student matrix whose entries are scores. Since question scores are typically nonnegative, it makes sense to apply NMF to this data to try to learn latent variables describing the relationship between students and questions. In the ideal view, these latent variables are skills [77]: the (i, j)-th entry of the feature matrix W is a relative measure of how much the i-th question tests the j-th skill, and the (j, k)-th entry of the weight matrix H represents the k-th student's strength at the j-th skill. Knowing this information would be useful to teachers, who would be better able to design tests to target specific skills as well as customize their teaching to individual students' skill sets, provided the skills correspond to interpretable, distinctive abilities. A diagram of this factorization is shown below:

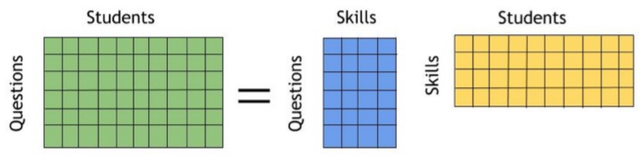


Figure 4.1: NMF for educational data

The question-by-skill matrix is also known as the Q-matrix [25], and will be the focus of our investigation here. The factorization and the nonnegativity constraints assume an additive skills model, in that a student's score on a question depends on the sum of the weights of the skills required times the student's aptitude at those skills. Each skill contributes a certain amount to an answer's success, and no skill is absolutely essential for success, unless it is the only skill tested by the question. This model contrasts to a conjunctive skills model, in which an answer can only be correct if the student has mastered *all* of the skills tested by the question [25].

In practice there are numerous complications to recovering the latent skills. For one, even if we assume an additive skills model, the factors that explain a student's testing performance may include the student's overall ability, the difficulty of particular questions, and other factors such as the student's health on a particular day and luck in guessing the right answer. In particular, the probability that a student knows the right answer but answers incorrectly is referred to as the slip probability, and the probability that the student does not know the answer but guesses correctly is known as the guess probability [5]. We would like to determine when the influence of these factors is weak enough, and the influence of skills is strong enough, for skill discovery through NMF to be possible. Secondly, educational datasets may have missing entries due to students not taking the full set of examinations. This is especially true for Massive Open Online Courses (MOOCs) in which the examinations are optional. Nevertheless, there are methods to reasonably estimate the missing entries [77]. We explain the missing entry estimation method we employ later. Lastly, recall that NMF is ill-posed, so there may exist multiple valid factorizations of a data matrix that correspond to different question-to-skills and skills-to-student mappings. However, over extensive testing we find that the NMF algorithms find very similar factorizations of the same data matrix, so this is not an issue.

The first complication is likely the most problematic in practice, as has been demonstrated by prior work. Winters et al. [77] were the first to apply NMF to educational data. They evaluated how well NMF could group questions from multiple educational datasets, ranging from undergraduate computer science test results to SAT Subject Test scores, by the skills the questions tested as given in the Q-matrix. They found that NMF performed no better than random clustering for sets of skills that were similar to each other, i.e. skills that could be covered in the same course, but it was effective for distinguishing questions that tested totally different skills, such as Mathematics and French, and concluded that using NMF to extract skills from educational datasets merited further investigation [77]. Desmarais [25] continued this study by building a synthetic model to attempt to determine whether the inaccurate recovery of similar skills in the tests by Winters et al. was due to the skewness towards high or low scores of the particular datasets they used, but found that the more likely reason was that skills were not a driving factor in the scores. In the last part of this section we will modify Desmarais' model and apply it to a new dataset structure to analyze under what conditions the Q-matrix may be recovered from the dataset. First, however, we share the results of NMF algorithms run on the real dataset.

4.1 Q-Matrix Recovery in Real Data

Our dataset contains binary quiz question results (correct, incorrect, or unanswered) for the students in the Coursera course *Networks Illustrated: Principles without Calculus* taught by Dr. Christopher Brinton and Prof. Mung Chiang (https://www.coursera.org/learn/networks-illustrated) [18]. The fact that the dataset is from one course, and its implication that the skills tested by the questions must be at least somewhat similar to each other, presents a challenge for Q-matrix recovery thru NMF. However, the pre-processed dataset has approximately three-quarters correct answers, much more than the Trivia and SAT datasets tested by Winters et al., which brings reason for optimism because we have seen that NMF algorithms are more accurate on less-sparse data. Furthermore, different teaching methods that vary across lectures, but not within lectures, may cater more to some students than others, which would mean that a question's lecture membership would be a driving force behind student performance on it. Finally, the low frequency of missing entries in the pre-processed dataset present another reason why we may be able to group the questions into lecture topics using NMF.

Indeed, we process the initially 70-by-3546 dataset to lower the count of and estimate the values of missing entries. Although we did construct a method to reasonably estimate missing entries, many of the students barely answered any questions at all, so it made sense to remove those students from the dataset entirely. On the other hand, there was a substantial group of faithful students who answered almost every question. Thus, we selected only the vectors from students who answered more than 60 of the 70 questions, of which there were 221 students. Finally, we removed the last question from our data because extremely few students answered it, leaving us with a 69-by-221 question-by-student matrix X_{NI}. For the remaining 3.45% of remaining entries that were missing, we estimated their value as:

$$X_{NI}(i,j) = \frac{1}{2} + \frac{15q_j + s_i - \dfrac{\nu}{2}}{p+n-2} \tag{4.1}$$

where q_j is the number of questions the j-th student answered correctly, s_i is the number of students who answered the i-th question correctly, ν is the number of

Questions	Lecture
1-10	"Power Control in Cellular Networks"
11-21	"Random Access in WiFi Networks"
22-29	"PageRank by Google"
30-38	"Pricing Data"
39-50	"Movie Recommendation on Netflix"
51-61	"Routing Traffic Through the Internet"
62-69	"Controlling Congestion in the Internet"

Table 4.1: Mapping of questions to Lecture Topics for the *Networks Illustrated* Coursera course.

questions answered by the j-th student plus the number of students who answered the i-th question, and p and n are the total numbers of questions and students, respectively. This formula provides an unbiased estimate of how likely the student is to have answered the question correctly based on the student's performance on other questions and the other students' performance on this particular question, weighted by the percent of questions the student answered and the percent of students who answered the question. Note that this estimate of missing entries does not assume any influence of a student's aptitude in particular skills involved in the question on the likelihood of a correct answer.

A visual depiction of the X_{NI} matrix is shown in Figure 4.2. Yellow pixels correspond to correct answers, blue pixels correspond to incorrect answers, and intermediate entries correspond to missing entries whose score has been estimated. The questions are broken up into seven distinct lectures; a map of questions to lectures is shown in Table 1.

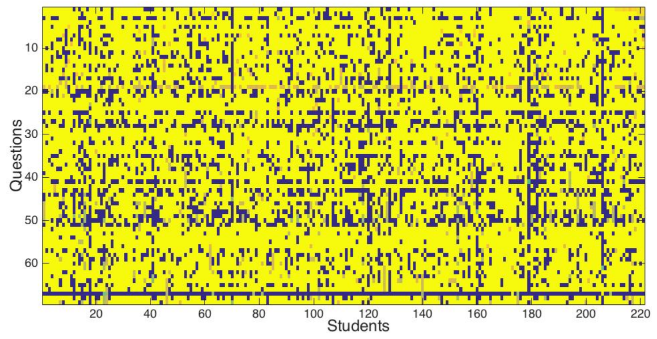


Figure 4.2: X_{NI} educational dataset processed from student score data for the *Networks Illustrated* Coursera course.

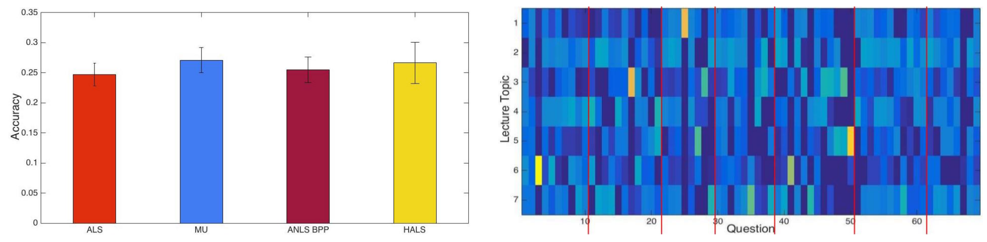
For our first test, we assume that these lecture topics are the latent skills. Our goal is to recover a 69-by-7 Q-matrix that accurately maps questions to the seven lecture topics. To do so, we compute a rank-7 NMF of X_{NI} and construct a contingency table from the resulting W matrix. The (i, j)-th element of the contingency table gives the number of questions from the i-th topic whose maximum element in the corresponding row in W was from the j-th column (symbolically, the j-th lecture topic). Essentially, this contingency table gives the empirical question clusters generated by NMF. To evaluate the accuracy of the Q-matrix, i.e. find the extent to which the empirical clusters match the ground-truth question to lecture topic clustering, we employ Desmarais' algorithm of permuting the rows of the contingency table to maximize the sum of diagonal elements, then dividing this sum by the total number of questions [25]. This algorithm accounts for the fact that NMF algorithms permute the columns of W corresponding to lecture topics, and ensures that each category of questions maps to a different lecture topic. An accuracy of 1 is perfectly accurate, and an accuracy of 0 is perfectly inaccurate.

Desmarais measured an accuracy of 0.72 for his SAT dataset and 0.35 for his trivia dataset (no better than random clustering after accounting for overfitting). Both datasets had 40 questions, 4 topics/skills and 100 examinees. When Desmarais trimmed the SAT dataset to only include the Mathematics and French topics, the yielded accuracy jumped to 0.96 [25]. Unfortunately, however, our results are much closer to those for the trivia data. Figure 4.3(a) shows the accuracy attained by factoring the X_{NI} dataset with different NMF algorithms, averaged over 20 trials with distinct random initializations[1]. MU achieves the best accuracy with a rate of 0.27, although the algorithms besides ALS all perform similarly. The algorithms do better than random clustering, which would label only approximately 1/7-th of the questions correctly, but not by much. Since MU, ANLS BPP, and HALS all tend to find a very similar Q-matrix, we choose MU to use for the rest of our tests in the interest of computing speed. Likewise, since the yielded Q-matrix does not vary across initializations, we use our default random initialization for all subsequent tests.

Figure 4.3(b) shows the transposed Q-matrix obtained by MU. The yellow entries are larger, the blue entries smaller, and the green entries somewhere in between. If the Q-matrix had accurately recovered the lecture topics, we would see that the maximum

[1]Note that this accuracy is not necessarily related to the relative error yielded by the NMF algorithms. Each of the four NMF algorithms tested yields a relative error of approximately 0.38 in factoring X_{NI}. Since X_{NI} has a mean of 0.76, this relative error is consistent with the relative error of 0.41 yielded by the NMF algorithms in computing a rank-10 NMF of a 50-by-250 binary matrix with likelihood of a zero entry equal to 0.25 (which we tested in Figure 3.12(a)).

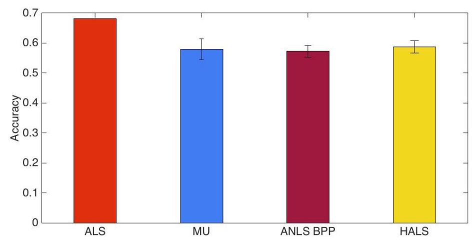
element of each question from the same lecture (delineated by the red lines) would belong to the same row (referring to the transposed Q-matrix as shown in the figure). Clearly this is not the case; the influence of the lecture topics is indiscernible. We will investigate the reasons for this in the next section.



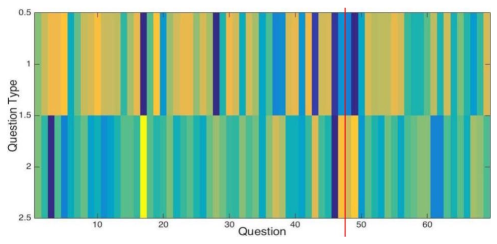
(a) (b)

Figure 4.3: (a) Lecture topic recovery accuracy averaged over 20 trials with displayed standard errors (b) Q-matrix generated by MU from Networks Illustrated dataset for $r = 7$

In an attempt to a more successful interpretation of latent variables thru NMF we next focused on the nature of individual questions as the latent variables instead of the lecture they belonged to. In particular, we divided each of the questions into two categories: definitional questions and application questions. The 48 questions in the former category were factual and knowledge-based; they asked students to recall some aspect of the definition of a concept. Conversely, application questions asked students to apply the definition of a concept to solve a problem they had likely never seen before. This is an oversimplification, but in general the types of thinking required to answer these two types of questions is undoubtedly different, so we presumed that the W matrix of a rank-2 NMF factorization of X_{NI} might designate questions by their definitional or applied nature. However, after permuting the questions in X_{NI} such that all questions of the same type were contiguous and repeating the same procedure described previously but now with 2 categories of questions instead of 7, the results were similarly inaccurate. As can be seen in Figure 4.4, the NMF algorithms performed no better than random clustering, and ALS yielded the highest accuracy only because it designated all of the questions as being of the same type.



(a)



(b)

Figure 4.4: (a) Question type recovery accuracy averaged over 20 trials with displayed standard errors (b) Q-matrix generated by MU from Networks Illustrated dataset for $r = 2$

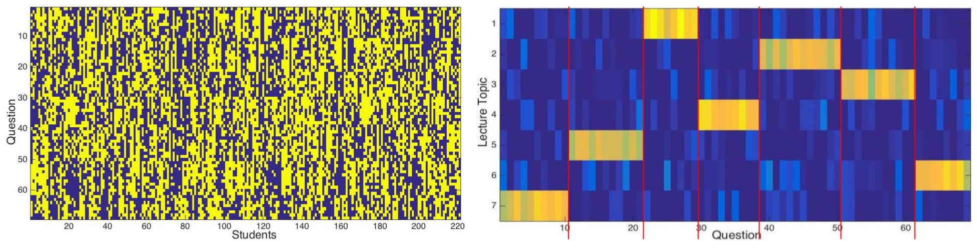
4.2 Simulations to Evaluate Feasibility of Recovering Q-Matrix

In this section we will adapt Desmarais' synthetic model to evaluate under what assumptions about the *Networks Illustrated* dataset it would be possible to correctly group the questions by lecture using NMF. In the interest of space, we do not apply this model to the question of recovering question types, though the same tests can be applied to that question with just a slight change of parameters.

4.3 Data Matrix Dependent Only on Lecture Topic

We start with the simplest, most ideal case, in which we assume that a question's lecture topic is the only factor that affects the likelihood of a correct score for a given student, and to do so we borrow Desmarais' model. Consider that each element x_{ij} of our synthetic data matrix $X \in \mathbb{R}^{69\times 221}$ is independently drawn from a Bernoulli distribution with parameter p_{ij}. This parameter is the same for each index belonging to the same lecture for the same student, and independent otherwise. In other words, each column of the matrix $P = [p_{ij}]_{1\leq i\leq 69, 1\leq j\leq 221}$ has 7 independent (and distinct) entries, one for each lecture. Each of the independent parameters are generated by a standard normal random variable z_{ij} that is then converted to a probability by computing the cumulative distribution function of z_{ij} for the standard normal distribution.

We observe in Figure 4.5 that this formulation allows for perfect clustering of questions by lecture through NMF. This occurs with high probability, as the clustering



(a) (b)

Figure 4.5: (a) Synthetic data matrix X generated under ideal conditions (b) Q-matrix generated by MU from X

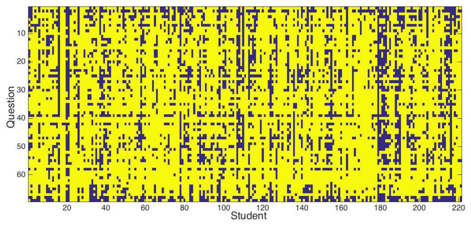
accuracy was always perfect for each of the many trials we conducted. Of course, the assumption that a question's lecture topic is the only factor that affects student success is unrealistic, so we next focus on the more interesting case where we account for additional factors.

4.4 Accounting for Question Difficulty and Overall Student Ability

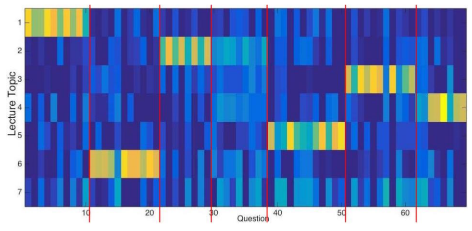
Question difficulty and overall student ability are arguably the two most obvious factors for student success on a question besides its lecture topic. The other major factor that comes to mind is random noise, representing conditions such as student health on a particular test day or luck in guessing the answer to a particular question, but we have already accounted for this factor to some extent by making the generation of X probabilistic. To account for question difficulty and overall student ability, we compute the intermediate variables z_{ij} indexed by the i-th question and j-th student as follows:

$$z_{ij} = a_{t_i,j} + b_j + c_i \tag{4.2}$$

where t_i is the lecture topic that the i-th question belongs to, and $a_{t_i,j}$, b_j, and c_i are Gaussian random variables with mean μ and standard deviations σ_t, σ_q, and σ_s, respectively. Each variable corresponds to the influence of lecture topic, item difficulty, and overall student ability in the correctness of an answer. We use different notation from Desmarais here in order to make it clear that each $a_{t_i,j}$ varies not only by the lecture topic t_i but also by the student j (since each student has a different aptitude for each lecture topic).



(a)



(b)

Figure 4.6: (a) Synthetic data matrix X generated with $\sigma_t, \sigma_q, \sigma_s = 1$ and $\mu = 0.5$ (b) Q-matrix generated by MU from X

The mean μ allows us to model the skewness of the data towards right or wrong answers. Desmarais implies that μ corresponds exactly to the mean of the data [25], but this is incorrect, since the cumulative distribution function is not linear. We could use a different μ for each lecture topic, but we use the same value here for simplicity. After generating all of the z_{ij} values, we compute the corresponding probabilities as the outputs of the standard normal cumulative distribution function of the z_{ij}s as we did previously.

Figure 4.6 shows the randomly generated data matrix X for $(\sigma_t, \sigma_q, \sigma_s) = (1, 1, 1)$ and $\mu = 0.5$ and the resulting Q-matrix W computed by MU. This instance represents the case when the influence of lecture topic, question difficulty, and student aptitude are all equal, and the mean of the data is approximately 0.75, as is the case for the X_{NI} dataset. Under these more realistic conditions, question clustering is no longer perfect, but it is still much better than random clustering. Over 40 trials, the clustering accuracy using MU has a mean of 0.78 and a standard deviation of 0.06.

We test additional parameter values and display our results in Tables 4.2 thru 4.6. For all of our tests we use the MU algorithm for NMF, since in all cases this algorithm yielded the best clustering accuracy or close to the best clustering accuracy compared to HALS, ANLS BPP, and ALS[2].

As we can see from Table 2, question clustering accuracy diminishes quickly as the lecture topic variance decreases relative to question difficulty variance and student ability variance. This is to be expected as the variance parameters determine how much each of the factors they represent affects students' answers. If each question's lecture topic has a relatively small influence on the correctness of each student's answer compared to other factors, we would expect that it would be more difficult

[2]Desmarais does not disclose which NMF algorithm he uses, which is somewhat problematic because we have seen that NMF performance varies across algorithms.

to recover the lecture topics from the student scores for each question, which Table 2 confirms.

Another reason for the diminishing question clustering accuracy is skewness of the data towards correct answers, as shown in Table 3. Interestingly, this comes even as the relative error of the nonnegative factorization diminishes with the increasing percentage of ones in X, suggesting that an NMF algorithm's accuracy is not necessarily related to its usefulness for certain applications. NMF relative error does not change significantly with the changing variances we use in our next tests, so we do not show the relative error in the NMF factorization for our subsequent tests.

Even more surprising is that question clustering accuracy increases as μ becomes more negative, until around $\mu = -0.5$. Interestingly, this is the most drastic increase in performance we found for when μ becomes negative; for assignments of parameters other than $(\sigma_t, \sigma_q, \sigma_s) = (1, 1, 1)$, the question clustering accuracy for negative values of μ is closer to or less than the accuracy at $\mu = 0$. We are unsure why the accuracy over μ behaves in this way; one rough conjecture is that negative values of μ happen to mask the affect of the question difficulty and student ability variance by making it more likely that they will contribute to a wrong answer regardless of how large their variance is. However, this conjecture does not account for the asymmetry in the behavior, i.e. why positive values of μ do not also mask the variances, so it remains unclear why the model behaves in this way.

Note that we could easily achieve better accuracy for tests such as $(\mu, \sigma_t, \sigma_q, \sigma_s) = (0.5, 1, 1, 1)$ by setting $X \leftarrow \text{NOT}(X)$, where NOT is the element-wise *not* operation, which would essentially change μ to $-\mu$. However, since this does not necessarily hold for other values of $(\sigma_t, \sigma_q, \sigma_s)$, and it is still interesting to know how the algorithms perform with $\mu = 0.5$ and no post-processing on X, we continue to use $\mu = 0.5$ (and not post-process X) for the rest of our tests. We did try the NOT trick on X_{NI}, but found that factoring X_{NI} yielded the same question clustering accuracy as factoring $\text{NOT}(X_{NI})$.

Contrary to what Desmarais claims [25], the question difficulty variance and overall student ability variance do not have symmetric effects on question clustering accuracy, as we deduce from Tables 4, 5, and 6. In each pair of tests across these tables where the question difficulty variance and student ability variance are swapped, the question clustering accuracy is always higher in the test where the student ability variance is larger. This makes sense because student ability contributes equally to student performance within each lecture topic, whereas question difficulty disrupts performance within each lecture topic, so question difficulty variance is more obstructive to clustering questions by lecture topic than is student ability variance.

Lastly, one might think that normalizing the rows or columns of X might help filter out the influence of μ and/or question difficulty or student ability, but this proved to not be the case - clustering accuracy actually decreased after normalization.

The most important takeaway from these results is that the lecture topic of questions from the *Networks Illustrated* course is likely not a significant factor in students' performance on that question. If it were, we would expect that we would be able to cluster the questions by lecture topic using NMF to an accuracy that is better than random clustering, but this is not the case. For example, even if lecture topic had only half the influence on student performance as both question difficulty and student ability, and if we account for the skewness of the data towards correct answers, we would expect to still be able to cluster the questions with an accuracy of 0.45 (as we see in Table 2), which is much better than the accuracy of 0.27 that we achieve in practice on the X_{NI} dataset. Indeed, the lack of influence of lecture topic should make sense to a large extent, because the lecture topics are similar in content and all delivered with essentially the same teaching style [18]. However, these results do not imply that lecture topics, or some other latent skills, cannot be recovered from other educational datasets in which the topics or skills are more distinct and presumably have more of an influence on student performance; we see this with Winters et al. and Desmarais' successful extraction of subjects from SAT Subject Test datasets. Likewise, the fact that we were unsuccessful in recovering lecture topics and question types from the X_{NI} dataset does not imply that other latent variables with greater influences may be successfully recovered from the X_{NI} dataset through NMF; it remains to be found what those potentially more influential latent variables are.

Table 4.2: Question clustering accuracy over varying β_t

μ	β_t	β_q	β_s	Mean Accuracy	Std. Dev. Accuracy
0.5	0	1	1	0.28	0.027
0.5	0.25	1	1	0.31	0.038
0.5	0.5	1	1	0.45	0.066
0.5	0.75	1	1	0.67	0.076
0.5	1	1	1	0.78	0.060
0.5	1.25	1	1	0.84	0.064
0.5	1.5	1	1	0.92	0.065

Table 4.3: Question clustering accuracy over varying μ

μ	β_t	β_q	β_s	Mean Accuracy	Std. Dev. Accuracy	Mean NMF Relative Error
-1	1	1	1	0.69	0.085	0.69
-0.75	1	1	1	0.85	0.087	0.62
-0.5	1	1	1	0.93	0.062	0.59
-0.25	1	1	1	0.91	0.063	0.53
0	1	1	1	0.88	0.065	0.45
0.25	1	1	1	0.84	0.071	0.41
0.75	1	1	1	0.67	0.076	0.26
1	1	1	1	0.54	0.085	0.18

Table 4.4: Question clustering accuracy over varying β_q with $\mu = 0.5$

μ	β_t	β_q	β_s	Mean Accuracy	Std. Dev. Accuracy
0.5	1	0	1	0.88	0.090
0.5	1	0.25	1	0.88	0.090
0.5	1	0.5	1	0.90	0.086
0.5	1	0.75	1	0.85	0.071
0.5	1	1.25	1	0.73	0.066
0.5	1	1.5	1	0.71	0.056

Table 4.5: Question clustering accuracy over varying β_s with $\mu = 0.5$

μ	β_t	β_q	β_s	Mean Accuracy	Std. Dev. Accuracy
0.5	1	1	0	0.84	0.090
0.5	1	1	0.25	0.83	0.076
0.5	1	1	0.5	0.81	0.086
0.5	1	1	0.75	0.79	0.058
0.5	1	1	1.25	0.77	0.064
0.5	1	1	1.5	0.78	0.058

Table 4.6: Question clustering accuracy over varying β_q and β_s with $\mu = 0$

μ	β_t	β_q	β_s	Mean Accuracy	Std. Dev. Accuracy
0	1	0	1	1	0
0	1	0.25	1	1	0
0	1	0.5	1	0.99	0.016
0	1	0.75	1	0.96	0.049
0	1	1	0	0.99	0.011
0	1	1	0.25	0.99	0.014
0	1	1	0.5	0.97	0.048
0	1	1	0.75	0.94	0.060

Chapter 5

Initialization Techniques

Since the NMF problem is nonconvex and the iterative algorithms that attempt to solve it follow descent directions of the objective function until they reach a local or global minimum (with the exception of ALS), the initial estimate given to these algorithms can play a critical role in determining the accuracy of the final solution. For example, consider the nonconvex objective function illustrated in Figure 5.1, and an iterative algorithm whose goal is to compute the point that minimizes the objective function. Assuming the algorithm computes a new point that is close to the previous point and in a descent direction on every iteration, if the initialization point is in the right valley, it will likely converge to the global minimum, but if it is in the left valley, then the algorithm will likely converge to the local minimum of that valley, which is less optimal than the global minimum. Note that ALS in the only NMF algorithm we have discussed so far that does not necessarily proceed in a descent direction on every iteration. For all the other algorithms, this picture implies that the initial estimate of the NMF solution can be highly influential in the accuracy of the final factorization.

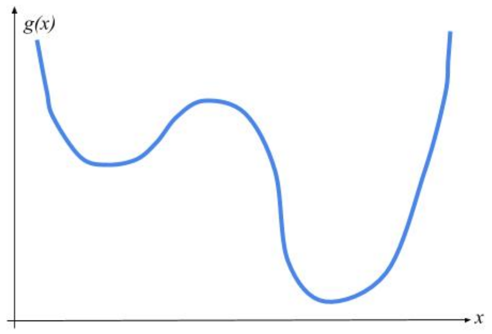


Figure 5.1: Generic nonconvex objective function $g(x)$.

Thus far, we have computed the initial factors $W^{(0)}$ and $H^{(0)}$ randomly by setting each of their elements to be equal to the absolute value of a random sample from the standard normal distribution. We refer to this initialization as the random initialization. In the remainder of this section, we introduce the reader to more methodical initializations, explaining the intuition behind them and the details of their execution. Like the heuristics they initialize, these initializations come with no performance guarantees in general, necessitating empirical evaluation of their ability to improve both the convergence rate and the final error of NMF heuristics, which we test on both synthetic and real data in the following section.

We do not cover all of the published initialization strategies here primarily because there are too many to cover each sufficiently in this space. Among the more prominent methods we leave out are independent component analysis [55] (a close relative of principal component analysis, which we do cover), Lanczos bi-diagonalization [74] (a technique motivated by initializing estimates for Orthogonal NMF, not the general NMF problem), Gabor wavelets [83] (a less popular technique based on the Gabor-wavelet transform), and population-based algorithms [51] (which must execute in parallel processes on distributed systems in order to make their accuracy worth their high cost). The initializations that we do cover can be categorized in three groups: data column subset selection techniques, clustering techniques, and low rank approximation techniques.

5.1 Data Column Subset Selection Techniques

The simplest NMF initialization methods involve computing $W^{(0)}$ by performing basic operations on subsets of the columns of X or the co-occurrence matrix XX^T, and computing $H^{(0)}$ according to the random initialization procedure. Because W then presumably has the more accurate initialization compared to H, the two-block coordinate descent algorithms first update H (compute $H^{(1)}$) before updating W when given an initialization of this type. The three methods of this type that we discuss here are all due to Langville et al. [57].

5.1.1 Random *Xcol*

The random *Xcol* initialization sets the columns of $W^{(0)}$ to be the average of a random subset of c of the columns of X [57]. Since the columns of W represent the basis vectors of the column space of X, this strategy hopes to accurately estimate the true basis vectors by taking random samples from the column space of X. If X is sparse, the random *Xcol* initialization seems to be an obvious improvement over the random initialization because the columns of $W^{(0)}$ will also be sparse. However, previous results have differed on whether the random *Xcol* initialization performs better than the random initialization [57, 21]. We test this conclusion for both sparse and non-sparse data later in this section. In our implementation, we set c equal to r, the number of columns of W. Thus, computing $W^{(0)}$ requires $O(r^2p)$ operations.

5.1.2 Random-C

The random-C initialization is a variant of the random *Xcol* initialization, but computes the columns of $W^{(0)}$ as the average of a random subset of only the largest in the ℓ_2-norm columns of X [57]. When X is very sparse, this method chooses columns that are the average of generally the densest columns of X, with the motivation being that these denser columns will be closer to the centers of clusters in the data [57]. For our implementation we compute each column of $W^{(0)}$ as the average over a random subset of r columns from the $4r$ largest columns of X (we assume $4r < n$). Accounting for norm calculation and sorting, computing $W^{(0)}$ requires $O(pn + n\log(n) + r^2p)$ operations.

5.1.3 Co-occurrence

The co-occurrence initialization method assigns the columns of $W^{(0)}$ to be the average of a random subset of columns among the largest columns (in the ℓ_2-norm) of the co-occurrence matrix XX^T [57]. This initialization requires a larger computational cost than the previous methods because of the cost to compute XX^T, but yields initial basis vectors contain information that the previous initial basis vectors do not, namely data point-to-data point similarities. We again average over a random subset of r columns from largest $4r$ columns (from XX^T this time) in our implementation. Computing $W^{(0)}$ thus requires $O(p^2n + r^2p)$ operations, including the cost to compute XX^T.

5.2 Clustering Techniques

The next set of initialization methods attempt to initially estimate the basis vectors of X by clustering the columns of X, then taking the centroids of those clusters. $H^{(0)}$ is then constructed based on some relation of the columns of X to the cluster centroids. The clustering may be *hard*, in which each data point belongs strictly to one cluster, or *soft*, in which each data point can belong to multiple clusters to varying degrees. By definition, the cluster centroids must be within the convex hull of the data, which is entirely within the nonnegative orthant, so clustering initialization techniques satisfy the nonnegativity constraints on $W^{(0)}$ (and $H^{(0)}$ as we will see).

As opposed to the column subset selection initialization methods, the clustering initializations entail that $W^{(0)}$ and $H^{(0)}$ have intuitive meanings that theoretically translate to favorable initializations. In particular, the fact that the columns of $W^{(0)}$ correspond to the average of clusters of the columns of X means that $W^{(0)}$ effectively represents the sparsity in X, which saves computational time during the NMF algorithm execution [76]. However, these clustering techniques are more expensive to compute than the previous methods. We test their effectiveness in practice in Chapter 6.

5.2.1 k-means

k-means is arguably the most well-known hard clustering algorithm. The goal of k-means clustering is to output a set of cluster centroids $\{c_j\}_{j=1}^k$ in $\mathbb{R}^p$, each corresponding to one of k clusters $\{\pi_j\}_{j=1}^k$ of columns of X [76]. For the purposes of initializing NMF, we set $k = r$, and set the r columns of $W^{(0)}$ to be the cluster centroids output by the k-means algorithm. The clusters are disjoint subsets, meaning that

$$\cup_{j=1}^k \pi_j = \{x_1, x_2, ..., x_n\} = X \quad \text{and} \quad \pi_i \cap \pi_j = \emptyset \forall i \neq j \tag{5.1}$$

Let $\Pi_k = \{\pi_j\}_{j=1}^k$ be a clustering of the n vectors into k clusters, and let $\iota(\Pi_k)$ be the length-n indicator vector associated with the clustering Π_k, where

$$\iota_i = j \text{ iff } x_i \in \pi_j \tag{5.2}$$

Next, let c_j be the centroid, or average, of the vectors in the j-th cluster. Since k-means attempts to minimize the distance of each vector to its cluster centroid (the *quantization error*), the natural objective function it tries to minimize is:

$$\Theta_{KM}(\Pi_k) := \frac{1}{2}\sum_{j=1}^{k}\sum_{x_i\in\pi_j} \text{dist}(x_i, c_j)^2 = \frac{1}{2}\sum_{i=1}^{n} \text{dist}(x_i, c_{\iota_i})^2 \tag{5.3}$$

where the ℓ_2-norm of the difference between each vector and its cluster centroid is typically used as the distance metric [76]. Not surprisingly, this objective is nonconvex in the cluster centroids. The problem of selecting the optimal partition for any reasonable clustering objective, including $\Theta_{KM}(\Pi_k)$, is NP-complete [76], so practical algorithms do not expect to find the strictly optimal clustering in general, but instead aim for a reasonably close-to-optimal solution.

There are several k-means algorithms that attempt to minimize Θ_{KM}. Most are iterative gradient descent procedures, and here we use the classical algorithm of this type presented by Bottou and Bengio in [13]. This algorithm updates the clusters on each iteration by assigning each vector x_i to the cluster associated with the closest centroid, then updates each centroid c_j according to the gradient descent rule:

$$c_j^{(t+1)} = c_j^{(t)} - \sigma \nabla_{c_j^{(t)}} \Theta_{KM} \tag{5.4}$$

where the step size σ is set to the multiplicative inverse of the number of vectors in π_j, and the gradient of Θ_{KM} with respect to c_j is

$$\nabla_{c_j^{(t)}} \Theta_{KM} = \sum_{x_i \in \pi_j} c_j - x_i \tag{5.5}$$

Since the objective function is nonconvex, the initialization for the k-means algorithm is very important, as a poor initialization may lead to the gradient descent method getting trapped in a highly suboptimal local minimum. To initialize the cluster centroids, we set them equal to distinct, random columns of X in order to prevent any cluster from have no members after any number of iterations. The entire k-means algorithm is outlined below.

Algorithm 5: k-means [76]

```
**Input:** Nonnegative matrix $X \in \mathbb{R}^{p \times n}$, integer $k$
**Output:** Nonnegative matrices $W^{(0)} \in \mathbb{R}^{p \times r}$ and $H^{(0)} \in \mathbb{R}^{r \times n}$, where $r = k$
  Initialize $k$ centroids:
  $\{c_j^{(0)}\}_{j=1}^{k} = \{x_{g_i}\}_{i=1}^{k}$, where $\{g_i\}_{i=1}^{k}$ is a set of distinct indices randomly chosen
  from the interval $[0, n]$
  $t \leftarrow 0$
  while Stopping condition not met do
      $t \leftarrow t + 1$
      Compute $\delta_{ij}^{(t)} = \|x_i - c_j^{(t-1)}\|_2$ for all $i = 1, 2, ..., n$ and $j = 1, 2, ..., k$
      Update each cluster: $\pi_j^{(t)} = \{x_i \mid \arg\min_l \delta_{il}^{(t)}\}$ for all $j = 1, 2, ..., k$
      Recompute each centroid:
          $c_j^{(t)} = c_j^{(t-1)} - \frac{1}{\mid \pi_j \mid}(\sum_{x_i \in \pi_j} c_j^{(t-1)} - x_i)$
  end while
  Compute $W^{(0)}$: $w_j^{(0)} = c_j$ for all $j = 1, 2, ..., k$
  See discussion in this section for computing $H^{(0)}$
```

This algorithm has been shown to converge to a local minimum of Θ_{KM} [76]. In fact, it does so superlinearly, meaning that the ratio of the error on the $(t+1)$-th iteration to the error on the t-th iteration approaches 0 as t goes to infinity [13]. However, there are no guarantees that this local minimum will be a global minimum.

The most commonly used method to compute $H^{(0)}$ is to set it equal to the solution to the NNLS problem $\arg\min_{H \geq 0} \|X - W^{(0)}H\|_F^2$ [76]. As we have seen in Section 2.4, there are multiple ways to solve this NNLS problem. However, in our implementation we use a different method of initializing H that entails projecting each x_i onto the space spanned by the centroids, and then setting the i-th column of $H^{(0)}$ equal to the coordinates of x_i in the centroid space. Namely,

$$h_{ji}^{(0)} = x_i^T \frac{c_j}{\|c_j\|_2^2} \text{ for all } j = 1, 2, ..., r; \quad i = 1, 2, ..., n \tag{5.6}$$

Note that after initializing W, $c_j \equiv w_j^{(0)}$, so either notation can be used in the above equation. Since both x_i and c_j are nonnegative, it follows that $h_{ji}^{(0)} \geq 0$ for all j, i, so this initialization $W^{(0)}, H^{(0)}$ is feasible. This technique is inspired by the method of initializing H mentioned in [21], in which each element of $H^{(0)}$ is set to be the distance from a corresponding column vector in X to its centroid. However, our method seems to make more intuitive sense when we consider that we want the product $W^{(0)}H^{(0)}$ to resemble the data matrix X, so if a column vector of X has a smaller distance to a centroid (column of $W^{(0)}$), we want that centroid to have a larger weight in our estimate of the corresponding column of X, not less. However, the projection method and the NNLS method yielded very similar results for k-means performance, so our choice to use the projection method does not make much of a difference.

Numerous stopping criteria may be defined for the k-means algorithm, including those based on changes in the objective function, the number of vectors changing clusters, and a previously specified maximum number of allowed iterations or minimum objective function value [76]. In our implementation we use a hard threshold of $\max(r, 5)$ iterations in the interest of simplicity and capping the computational investment we are allowed to spend on initializing the NMF algorithms. In all of our testing of the k-means algorithm (and the spherical k-means algorithm) over various datasets, $\max(r, 5)$ iterations were sufficient to achieve convergence of the objective function.

Each iteration of k-means costs $O(prn)$ operations to compute the distances and new centroids, which is equal to the cost per iteration of most of the NMF heuristics, though the k-means procedure has a smaller constant [76]. Nevertheless, this raises the question of whether it would be more effective to spend the time running k-means instead by simply running the NMF heuristics with a random or cheaper initialization. We run experiments to attempt to answer this question in Chapter 6.

Next, however, we must point out a major flaw with k-means. Often in real applications we care more about the directions of the data points than their magnitude. For example, in topic modelling, we would like to put two documents that have the same frequencies of words but a different total number of words in the same cluster, while we would like to put a document that has a different word frequency portfolio but a similar total number of words in a separate cluster. In other words, we would like our cluster centroids to be linearly independent. The same applies for initializing NMF - the columns of $W^{(0)}$ should be linearly independent so that they may capture more of the column space of X. Yet k-means clusters data points based on their locations in p-dimensional space, so if the points are not normalized, it clusters them according to both their magnitude and direction. Thus, cluster centroids may be linearly dependent if their members have similar directions but different magnitudes. The next initialization technique addresses this flaw in k-means.

5.2.2 Spherical k-means

The spherical k-means method is very similar to the k-means method, but first normalizes the columns of X such that they all have unit Euclidean norm before clustering them. Originally proposed by Dhillon and Modha in 2001 [27] and later developed by Wild [76], the method's normalization step weighs each point equally in the dataset, and thereby accounts for only its direction and not its magnitude. Because of this, spherical k-means yields centroids that are more linearly independent than those yielded by k-means [76].

Due to the nonnegativity of the data, each normalized data point lies on the unit hypersphere in the positive orthant of $\mathbb{R}^p$. A two-dimensional case is shown in Figure 5.4b. This restriction allows us to quantify the similarity between two data points x_i and x_j as a function of the angle between them using the cosine similarity measure:

$$\cos(\theta_{x_i,x_j}) = \|x_i\|_2 \|x_j\|_2 \cos(\theta_{x_i,x_j}) =: x_i^T x_j \tag{5.7}$$

where the first equality follows from the normalization of x_i and x_j, and the second follows from the definition of the inner product. The larger the angle θ_{x_i,x_j} between the two vectors (or equivalently, the larger the positive difference between their directions), the smaller $\cos(\theta_{x_i,x_j})$ will be. Since x_i and x_j are confined to the positive orthant, $\cos(\theta_{x_i,x_j}) \in [0, 1]$, and values of the cosine similarity measure closer to one correspond to more similar points while values closer to zero correspond to less similar points. We can compute the cosine similarity measure using the inner product, which is computationally cheaper than computing the Euclidean distance, as is done in the k-means algorithm, especially if either of the vectors are sparse, excluding normalization costs.

To use the cosine similarity measure in the spherical k-means algorithm, the centroids must be normalized throughout the algorithm [76]:

$$c_j = \frac{\hat{c}_j}{\|\hat{c}_j\|} = \frac{\sum_{x_i \in \pi_j} x_i}{\|\sum_{x_i \in \pi_j} x_i\|} \tag{5.8}$$

The coherence $Q(\pi_j)$ of a cluster π_j with centroid c_j can now be defined as [76]:

$$Q(\pi_j) \coloneqq \sum_{x_i \in \pi_j} x_i^T c_j \tag{5.9}$$

Using the Cauchy-Schwarz Inquality, one can show that the choice of c_j in 5.8 maximizes $Q(\pi_j)$ [76]. This result confirms that c_j is the closest vectors to all the vectors in a the corresponding cluster.

Since the goal of spherical k-means is to minimize the distance between data points and their centroids, and minimizing the total distance between a centroid and its cluster points is analogous to maximizing the coherence of the cluster, the objective function of spherical k-means can be written as the sum of all the cluster coherences, which the spherical k-means algorithm will try to maximize [76]:

$$\Theta_{SKM}(\Pi_k) = \sum_{j=1}^{k} Q(\pi_j) = \sum_{j=1}^{k} \sum_{x_i \in \pi_j} x_j^T c_i = \sum_{i=1}^{n} x_i^T c_{\iota_i} \tag{5.10}$$

We use use the spherical k-means algorithm as it is presented by Wild [76]. This algorithm follows a similar structure as the k-means algorithm, but does not update the centroids according to a gradient descent (or because of the goal of maximization, gradient ascent) rule. Instead, it recomputes the centroids as the average of the vectors in their cluster after re-clustering the vectors on each iteration.

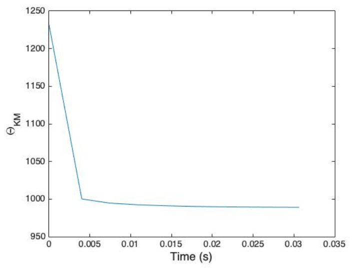
Algorithm 6: Spherical k-means [76]

```
Input: Nonnegative matrix $X \in \mathbb{R}^{p \times n}$, integer $k$
Output: Nonnegative matrices $W^{(0)} \in \mathbb{R}^{p \times r}$ and $H^{(0)} \in \mathbb{R}^{r \times n}$, where $r = k$
  Normalize the columns of $X$: $x_i = \frac{x_i}{\|x_i\|_2}$ for all $i = 1, 2, ..., n$
  Initialize $k$ centroids:
  $\{c_j^{(0)}\}_{j=1}^{k} = \{x_{g_i}\}_{i=1}^{k}$, where $\{g_i\}_{i=1}^{k}$ is a set of distinct indices randomly chosen from the interval $[0, n]$
  $t \leftarrow 0$
  while Stopping condition not met do
      $t \leftarrow t + 1$
      Compute $\rho_{ij}^{(t)} = x_i^T c_j^{(t-1)}$ for all $i = 1, 2, ..., n$ and $j = 1, 2, ..., k$
      Update each cluster: $\pi_j^{(t)} = \{x_i \mid \arg\max_l \rho_{il}^{(t)}\}$ for all $j = 1, 2, ..., k$
      Recompute each centroid:
          $$c_j^{(t)} = \frac{\sum_{x_i \in \pi_j^{(t)}} x_i}{\| \sum_{x_i \in \pi_j^{(t)}} x_i \|_2}$$
  end while
  Compute $W^{(0)}$: $w_j^{(0)} = c_j$ for all $j = 1, 2, ..., k$
  Compute $H^{(0)}$ (many options for this, we use the method described in Section 5.2.1).
```

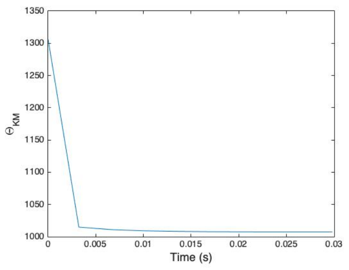
This algorithm has been proven to be nondecreasing in the objective function Θ_{SKM} [27]. It is also guaranteed to converge to a local maximum of Θ_{SKM}, but has no guarantees on achieving a global maximum due to the nonconvexity of Θ_{SKM} [76]. Dhillon et al. proposed a modification to the spherical k-means algorithm to allow it to escape local maxima and help it find the global maxima [26], but this modified algorithm is computationally more expensive and similarly is not guaranteed to find a global maximum.

Like k-means, one benefit of spherical k-means is that it preserves the sparsity of X in its initial estimate of W. Another benefit is that it generates linearly independent centroids, which k-means does not necessarily do. Furthermore, although spherical k-means still requires $O(prn)$ operations per iteration to compute both $\rho_{ij}^{(t)}$ and $c_j^{(t+1)}$ for all i, j, the constant associated with this number of operations is smaller than it is for k-means, since it is cheaper to compute the similarity $\rho_{ij}^{(t)}$ via the inner product ($2p$ operations) than it is to compute the distance $\delta_{ij}^{(t)}$ via the Euclidean norm of the difference between two vectors ($3p$ operations).

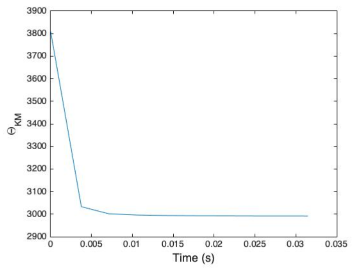
In order to understand the behavior of k-means and spherical k-means irrespective of their effectiveness as NMF initialization methods, we run three basic tests in Figures 5.2 and 5.3. Figure 5.2 shows the values of Θ_{KM} over time for three different datasets X, where X is always clustered into $r = k = 10$ clusters and has $p = 50$ rows and $n = 250$ columns. The algorithm runs for $\max(r, 5) = 10$ iterations, and each test is averaged over 10 trials. In Figure 5.2a, each entry of X is the absolute value of an independent sample from the standard normal distribution. In Figure 5.2b, each entry of X is initially generated as in in Figure 5.2a, then constant 5 is added to each entry of the last 125 columns of X. Likewise, in Figure 5.2b each entry of X is generated as in Figure 5.2a, then each entry of the last 125 columns of X are multiplied by 5. We run the analogous tests in Figure 5.3, but on the spherical k-means algorithm instead of the k-means algorithm.



(a)



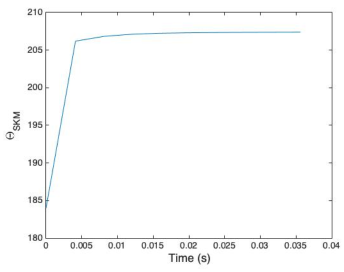
(b)



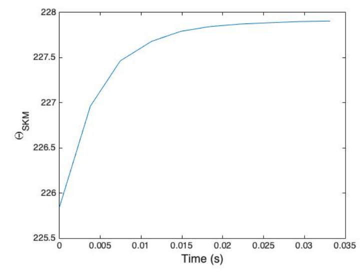
(c)

Figure 5.2: Value of k-means objective function over time for 10 iterations, where $X \in \mathbb{R}^{50x250}$, $k = 10$, and in (a) $x_{ij} \sim$ |N(0,1)|, (b) half the data is shifted, and (c) half the data is scaled.

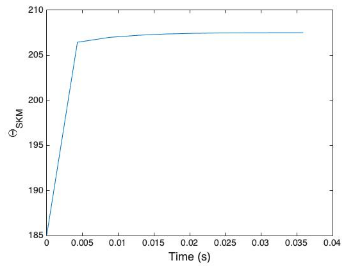
Both algorithms converge in just a few iterations in each case, and they always increase or decrease their respective objective function monotonically. Also observe that k-means takes shorter time to run than spherical k-means, which contradicts our



(a)



(b)



(c)

Figure 5.3: Value of spherical k-means objective function over time for 10 iterations, where $X \in \mathbb{R}^{50\times 250}$, $r = k = 10$, and in (a) $x_{ij} \sim$ |N(0,1)| (b) half the data is shifted, and (c) half the data is scaled.

previous analysis. However, this is most likely due to overhead normalization costs outweighing the reduced time per iteration of spherical k-means for these relatively small values of p, r, and n, since we find that for larger values of p, r, and n, spherical k-means runs faster than k-means.

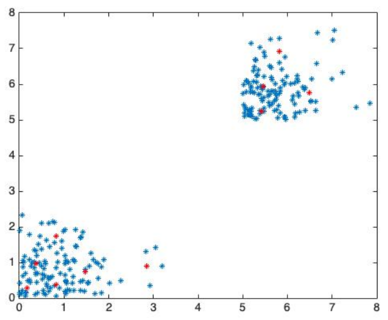
Using Figure 5.2a as a baseline, we see that spherical k-means performance is roughly the same in Figure 5.2b. We would expect to obtain more meaningful clustering in this situation, since half of the data points in X are shifted in p-dimensional space, creating two large clusters of points. k-means is able to take advantage of this in the sense that it splits its centroids into two groups associated with each of the natural clusters (see the visualization of the simplified case when $p = 2$ in Figure 5.4a), but its objective function does not decrease from Figure 5.2a because Θ_{KM} increases with the as the norms of the data points in X increase. This is why we see the final value of Θ_{KM} increase three-fold when half the columns of X are positively scaled in Figure 5.2c.

On the other hand, spherical k-means is invariant to scaling of the data, as we see in the near equivalence of Figure 5.3c to Figure 5.3a. Furthermore, spherical k-means improves its objective function value when half the data is shifted. Although this shifting does not create two separate clusters in p-dimensional space in the eyes of

Table 5.1: $\kappa(W^{(0)})$ for $W^{(0)}$ returned by k-means and spherical k-means for the tests in Figures 5.2 and 5.3

	(a) Baseline data	(b) Half shifted data	(c) Half scaled data
k-means	17.97	151.58	119.94
Spherical k-means	15.55	31.64	15.38

the spherical k-means algorithm, it does give the distribution of data points on the p-dimensional hypersphere additional structure than the essentially uniform distribution in the other cases, which the algorithm takes advantage of. From an optimization standpoint, this additional structure perhaps reduces the number of local minima of Θ_{SKM}, potentially explaining why spherical k-means increases Θ_{SKM} for more iterations than in the other cases before converging.


(a) k-means

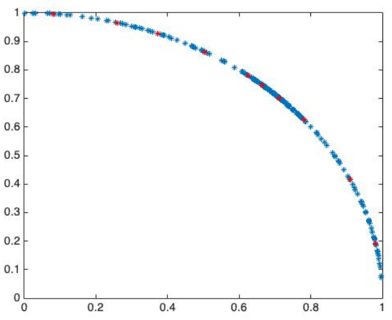

(b) Spherical k-means

Figure 5.4: Centroids (shown in red) generated by (a) k-means and (b) spherical k-means for the same data points (shown in blue) up to normalization, for the half-shifted data case.

Figures 5.4a and 5.4b show the centroids generated by k-means and spherical k-means for the same tests as in Figures 5.2b and 5.3b, respectively, but with $p = 2$ in each case. The centroids that spherical k-means generates are preferable because of their linear independence; this is also reflected in the condition numbers of $W^{(0)}$ reported in Table 5.1 for the tests of k-means and spherical k-means. k-means generates centroids that are nearly linearly dependent when enough data points are nearly scaled versions of each other, so the matrix of centroids $W^{(0)}$ has a large condition number in these situations, but spherical k-means always generates linearly independent centroids, so its returned $W^{(0)}$ is always well-conditioned. These results support our prior theoretical reasoning.

Finally, note that although spherical k-means, and potentially k-means, may provide a suitable clustering of the data, the clustering centroid basis may not be the best basis to represent the columns of X for the purposes of nonnegatively factoring X. Despite the extra computational costs of clustering, the centroid basis may not allow for the linear combinations of it to best approximate all of the columns of X, and may potentially lead the NMF algorithms to converge to a highly suboptimal local minimum [76]. These concerns are also relevant to the next clustering algorithm we discuss, Fuzzy C-means, and motivate our testing of these initializations in Chapter 6.

5.2.3 Fuzzy C-means

Fuzzy C-means is the most popular soft clustering algorithm [30]. As such, unlike the two previous hard clustering algorithms, fuzzy C-means allows data points to split their membership among multiple clusters. To the best of our knowledge, this paradigm was first proposed by Bezdek in 1981 [10]. It allows for better modelling of situations in which the data points should not be strictly segregated into distinct clusters, but belong to multiple clusters at once. We will test whether this model better suits the task of computing a representative low-dimensional basis for the column space of the data X when we test its effectiveness as an NMF initialization method in Chapter 6.

The fuzzy C-means objective function models the soft clustering problem as follows [30]:

$$\Theta_{FCM} = \sum_{i=1}^{n} \sum_{j=1}^{k} u_{ij}^{m} \|x_i - c_j\|_2^2 \tag{5.11}$$

where u_{ij} is the portion of the data point x_i's membership that belongs to the cluster j, and $m > 1$ is a weighting exponent for the degree of fuzziness. The total membership that each data point is able to distribute among clusters is 1, so we have the constraints that

$$\sum_{j=1}^{k} u_{ij} = 1 \text{ for all } i = 1, 2, ..., n \tag{5.12}$$

$$u_{ij} \in [0, 1] \text{ for all } i = 1, 2, ..., n \text{ and } j = 1, 2, ..., k \tag{5.13}$$

$$\tag{5.14}$$

Another distance metric besides the ℓ_2-norm of the difference between the two vectors may be used in the fuzzy C-means objective function [30], but in our implementation we again use the 2-norm of the difference because it is a natural distance metric and for simplicity and consistency. $m = 2$ is the most common choice of m, and the value we use.

The fuzzy C-means algorithm uses an alternating minimization procedure to minimize Θ_{FCM} subject to the constraints. In particular it, updates the membership weights u_{ij} and the cluster centroid c_j on each iteration as follows [30]:

$$u_{ij} = \frac{1}{\sum_{l=1}^{k} \left(\frac{\delta_{ij}}{\delta_{il}}\right)^{2/(m-1)}} \tag{5.15}$$

$$c_j = \frac{\sum_{i=1}^{n} u_{ij}^m x_i}{\sum_{i=1}^{n} u_{ij}^m} \tag{5.16}$$

where again k is the number of clusters, and $\delta_{ij} = \|x_i - c_j\|_2$ is the distance between the i-th data point and the j-th centroid defined by the Euclidean norm of the difference between the two vectors.

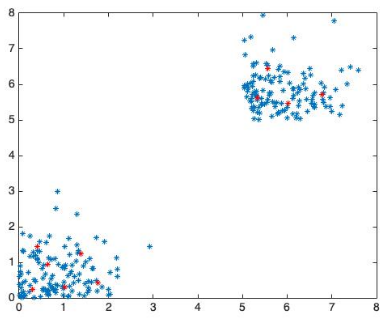
Algorithm 7: Fuzzy C-means [30]

```
**Input:** Nonnegative matrix $X \in \mathbb{R}^{p \times n}$, integer $k \leq \min(p, n)$, integer $m$
**Output:** Nonnegative matrices $W^{(0)} \in \mathbb{R}^{p \times r}$ and $H^{(0)} \in \mathbb{R}^{r \times n}$, where $r = k$
  Initialize $k$ centroids using Random $Xcol$
  $c_j^{(0)} = \{x_{g_i}\}_{i=1}^{k}$, where $\{g_i\}_{i=1}^{k}$ is a set of distinct indices randomly chosen from the interval $[0, n]$
  $t \leftarrow 0$
  **while** Stopping condition not satisfied **do**
      $t \leftarrow t + 1$
      Compute $\delta_{ij}^{(t-1)} = \|x_i - c_j^{(t-1)}\|_2$ for all $i = 1, 2, ..., n$ and $j = 1, 2, ..., k$
      Update $u_{ij}^{(t)}$ according to 5.15 for all $i = 1, 2, ..., n$ and $j = 1, 2, ..., k$
      Update $c_j^{(t)}$ according to 5.16 for all $j = 1, 2, ..., k$
  **end while**
  Set $w_j^{(0)} = c_j$ for all $j = 1, 2, ..., k$
  Set $H^{(0)} = U^T$
```

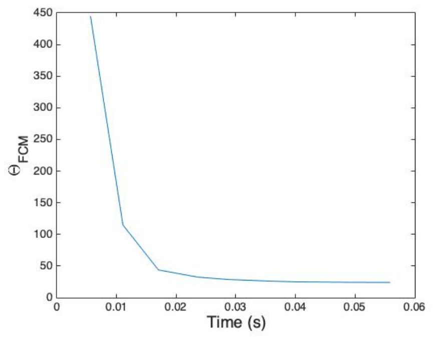
The entire fuzzy C-means algorithm is shown in Algorithm 7. We use the random *Xcol* method to initialize the centroids because setting the centroids equal to random columns of X would cause some δ_{ij} to be 0 and a division by 0 on the first iteration. There is no need to find the closest cluster for each data point to determine its membership, since its membership is split across all clusters. Computing each of the δ_{ij} values before computing the u_{ij} values saves substantial computational cost from having to recompute them multiple times if we computed the u_{ij} values first, as is suggested by the versions of the algorithm presented elsewhere [29]. The algorithm presented here requires approximately $2prn$ operations to compute all of the δ_{ij} values, $2prn$ operations to compute all of the u_{ij} values, and $3pn$ operations to compute all of the centroids. This makes for a total of $O(prn)$ operations per iteration, equal to the previous clustering algorithms and most of the NMF heuristics.

For stopping conditions, we impose a hard threshold of r on the number of iterations t in order to remain consistent with the previous clustering algorithms and save computational cost per iteration. In all of our empirical tests, this threshold was sufficient for convergence.

To exemplify fuzzy C-means behavior we show in Figure 5.5 the final centroids computed by the algorithm for the half-shifted, two-dimensional data we generated in the previous section, and the evolution of Θ_{FCM} on this same data over time for 10 iterations. The value of Θ_{FCM} on the first iteration is not shown because U has yet to be initialized at that point. Note that the objective function value decreases monotonically and converges in just a couple iterations. Also, fuzzy C-means generates 6 centroids in the closer of the two general clusters to the origin, whereas k-means split the centroids evenly: 5 centroids in the general cluster closer to the origin, and 5 in the further general cluster.



(a)



(b)

Figure 5.5: (a) centroids (shown in red) generated by fuzzy C-means for the data points in blue (b) evolution of Θ_{FCM} over time for 10 iterations on the same data

Perhaps the largest intuitive advantage to fuzzy C-means clustering comes from the natural generation of $H^{(0)}$. Since $h_{ji}^{(0)} = u_{ij}$, the (j, i)-th element in $H^{(0)}$ gives the degree to which the i-th data point belongs to the j-th cluster. Thus, if the i-th data point belongs heavily to the j-th cluster, the j-th centroid (j-th column of $W^{(0)}$) will be weighted heavily in the estimation of the i-th data point, $x_i \approx W^{(0)} h_i^{(0)}$, which makes sense in our effort to produce an accurate estimation. We will soon discuss how well this works in practice in Chapter 6.

5.2.4 Subtractive Clustering

Subtractive clustering is another soft clustering algorithm, but it follows a different structure than the previous clustering algorithms in the sense that it never explicitly computes each data point's membership in clusters during centroid computation, and it greedily selects cluster centroids without ever updating them once they have been selected. The subtractive clustering method has been known since at least 1994 [22], but it was not until 2013 that Casalino et al. proposed it as an NMF initialization scheme [21].

Like the previous clustering algorithms, subtractive clustering tries to compute centroids that form a basis that accurately represents the column space of X. To do so, it greedily selects cluster centroids from the data points in X as those that are most likely to be cluster centroids based on the distances of nearby data points [21]. In particular, it fist normalizes each data point such that it has unit Euclidean norm and only its direction is considered, then assigns each of these data points x_i a potential p_i, defined as:

$$p_i = \sum_{j=1}^{n} \exp\Big(-\frac{4}{r_a^2} \|x_i - x_j\|_2^2 \Big) \text{ for all } i = 1, 2, ..., n \tag{5.17}$$

where r_a is a positive constant dictating how much to weigh nearby points [21]. Since points with smaller distances to the other points have higher potentials by the definition above, the algorithm sets the point x_j with the highest potential $\tilde{p}_j$ to be the first cluster centroid. Next, to account for the fact that this cluster has already been chosen, and to mitigate the likelihood that a nearby point will later be chosen as a cluster centroid, the algorithm subtracts an amount proportional to the most recently chosen centroid from each data point's potential as follows [21]:

$$p_i \leftarrow p_i - \tilde{p}_j \exp\Big(-\frac{4}{r_a^2} \|x_i - x_j\|_2^2 \Big) \text{ for all } i = 1, 2, ..., n \tag{5.18}$$

where r_b is also a positive constant . The algorithm next selects the data point with the highest remaining potential to be the next cluster centroid, and iterates this process until r centroids have been selected [21].

The columns of $W^{(0)}$ are set to be the cluster centroids in the order that they were selected, such that $w_1^{(0)}$ is the first centroid that was selected and $w_r^{(0)}$ is the last. As is the case with the previous clustering algorithms, the elements of $H^{(0)}$ are set to be measures of how much each data point belongs to the cluster associated with each centroid. Namely, the algorithm computes $h_{ij}^{(0)}$ as follows [21]:

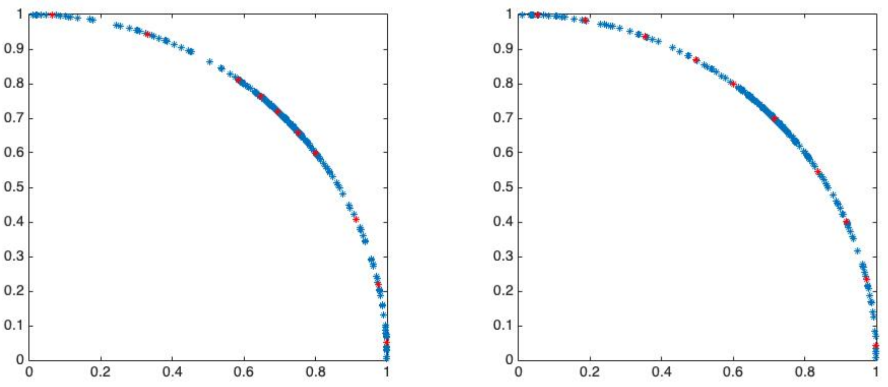
$$h_{ji}^{(0)} = \frac{\exp\Big(-\dfrac{1}{2}\dfrac{\|x_i - w_j^{(0)}\|_2^2}{\sigma^2}\Big)}{\sum_{l=1}^{r} \exp\Big(\dfrac{\|x_i - w_l^{(0)}\|_2^2}{\sigma^2}\Big)} \tag{5.19}$$

where $\sigma^2 = \dfrac{r_a^2}{8}$. The denominator represents a normalization on each column of $H^{(0)}$ to ensure that $W^{(0)}H^{(0)}$ is properly scaled with X. Note that this algorithm can also be used to select an appropriate value of r, by modifying it such that it selects centroids until the maximum potential is less than some threshold, typically $0.15\tilde{p}_j$ [21].

Subtractive clustering depends on two parameters, r_a and r_b. The former weighs the influence of surrounding points such that nearby points are weighted more heavily. For instance, if a point is further than r_a from another point, the potential each point contributes to the other's potential is less than $\exp\big(-\dfrac{4}{r_a^2}r_a^2\big) = e^{-4} \approx 0.183$. Thus, r_a can be interpreted roughly as the minimum distance from a centroid within which a point may be considered a member to a non-trivial degree of the centroid's cluster [21]. After tuning, we choose $r_a = 0.1$ in our implementation.

Similarly, r_b provides neighborhood within which points may be considered members in a centroid's cluster. This time, it is used to reduce the potential of points close to a centroid that has just been selected. Since we do not want select a centroid that belongs to another centroid's cluster, or even to have two clusters to overlap significantly, we must choose r_b to be at least as large as r_a . Indeed, $r_b = 1.5r_a$ is the typical choice used in the literature [21], and the choice we use in our implementation.

The algorithm's computational efficiency can be improved by first computing each squared distance $\|x_i - x_j\|_2^2$, which can be done in approximately $3pn^2$ operations. Afterwards, computing all of the initial potentials requires just $3n$ operations, and updating all of the potentials requires $4n$ operations for each round of updates. Since there are only r rounds of updates, and the initial normalization takes $2pn$ operations, the entire asymptotic computational cost is $O(pn^2 + rn)$ operations, which may be cheaper than the $O(prn\times$ the number of iterations) operations required by the other clustering intializations if n/r is not larger than the number of iterations required by the other methods. Importantly, the dominant cost from the algorithm is computing the distances, which can be done once per dataset, so the algorithm can be run many times on a dataset very quickly to tune r_a and r_b [21]. Lastly, the computational cost to compute $H^{(0)}$ is $O(prn)$ operations.



(a) Centroids generated by conventional subtractive clustering algorithm.

(b) Centroids generated by modified subtractive clustering algorithm with cosine similarity measure.

Figure 5.6

Since the data points are normalized, computational time may be reduced slightly, from $3pn^2$ to $2pn^2$ operations, by using the cosine similarity measure introduced in the context of spherical k-means as a 'distance' metric instead of the Euclidean norm of the difference between two vectors. However, this would require changing the algorithm elsewhere to account for closer points now having larger 'distance' measures. To account for this we adjust each of the exponential terms as follows:

$$\exp\Big(-\frac{4}{r^2_{(a \text{ or } b)}}\|x_i - x_j\|_2^2\Big) \to \exp\Big(\frac{4}{r^2_{(a \text{ or } b)}}(x_i^T x_j - 1)\Big) \tag{5.20}$$

so that a larger potential still corresponds to more points being nearby, and the maximum attainable value of the expression is 1 (which causes the potential of a point to become 0 after the point becomes a centroid). We show the centroids calculated by the conventional version of the algorithm and those calculated by the modified cosine similarity version in Figure 5.6. Both of these sets of centroids appear to be reasonable. In the interest of space, and since the modified version saves relatively little computational cost, we only test the conventional subtractive clustering algorithm's efficacy as an initialization method in Chapter 6.

5.3 Low Rank Approximation Techniques

Low rank approximation initializations construct $W^{(0)}$ and $H^{(0)}$ based on the eigenstructure of the data. Whereas clustering algorithms may be viewed as geometric 'low rank' approximation algorithms, the low rank approximation algorithms use the term 'low rank' in the algebraic sense. They have similarly expensive computational costs to the clustering algorithms, and like the clustering algorithms, have been shown to provide significant performance boosts to NMF algorithms in terms of increasing their convergence rate and ultimate error [16].

5.3.1 SVD-Centroid

We start by discussing a notable low rank initialization, the SVD-Centroid procedure [21, 57], that we do not implement due to concerns over its feasibility. The algorithm's authors describe its procedure as using the centroid decomposition of the SVD factor V (we will discuss the SVD in greater detail later) to initialize W [57]. By the centroid decomposition of a matrix, they refer to the centroids generated one of the clustering algorithms discussed previously. The motivation for this technique is that $V \in \mathbb{R}^{n\times k}$ is smaller than $X \in \mathbb{R}^{p\times n}$, so clustering its columns will be less computationally expensive than clustering the columns of X. Yet the columns of V represent the row space of X and are n-dimensional, whereas the columns of of $W^{(0)}$ should represent the column space of X and must be p-dimensional. If instead we were to set the centroids of V to be the rows of $W^{(0)}$ in hopes that $W^{(0)}$ would then emulate the row space of $W^{(0)}H^{(0)} \approx X$, this would again not make sense because the rows of X are not linear combinations of the rows of $W^{(0)}$. Furthermore, even if we were to adjust the algorithm to make more geometric sense, such as taking the centroid decomposition of the SVD factor U, whose columns represent the column space of X, we would still need to account for the negative elements in U or V, which the authors provide no instruction on how to do [57]. For these reasons, we seek greater clarification on the SVD-Centroid technique before implementing it.

5.3.2 PCA

The first low rank approximation initialization algorithm we implement is based on principal component analysis (PCA). PCA is a well-known low rank approximation algorithm, and was modified for use as an NMF initialization method by Zhao et al. in [82]. The method first executes computes the first r principal components (directions of the most variance) of the data matrix X using PCA. To do so, it first computes the covariance matrix $C \in \mathbb{R}^{p \times p}$ of the data $X \in \mathbb{R}^{p \times n}$:

$$C = \sum_{i=1}^{n} (x_i - \mu)(x_i - \mu)^T \tag{5.21}$$

where

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{5.22}$$

is the mean of the data points x_i [82]. We can also define the centered data matrix $A = [x_1 - \mu, x_2 - \mu, ..., x_n - \mu]$, which allows us to express C as $C = AA^T$. Next, the method computes the eigenvectors $u_1, u_2, ..., u_p$ and eigenvalues $\lambda_1, \lambda_2, ..., \lambda_p$ of C. The eigenvectors are the principal components, and their corresponding eigenvalues represent the variance of the data in the direction of the eigenvector.

Once the principal components have been computed, the NMF initialization algorithm selects the r components with the largest corresponding eigenvalues and sets the columns of $W^{(0)}$ to equal to these components [82]. The algorithm then computes the columns of $H^{(0)}$ via $h^{(0)} = W^T(x_i - \mu)$ for all $i = 1, 2, ..., n$. Finally, the algorithm post-processes $W^{(0)}$ and $H^{(0)}$ by updating them as follows: $W^{(0)} \leftarrow \min(1, |W^{(0)}|)$ and $H^{(0)} \leftarrow \min(1, |H^{(0)}|)$, where min is the element-wise minimum operation. This post-processing ensures that each of the entries of $W^{(0)}$ and $H^{(0)}$ are in the range $[0, 1]$, which satisfies the nonnegativity constraints and provides a conservative scaling of the entries, but the absolute value operation and truncation obscures the underlying meaning of the entries [82]. We evaluate how this method performs in practice in Chapter 6.

Besides computing the eigenvalues and eigenvectors of AA^T, the initialization scheme requires $O(prn + p^2 n)$ operations. Unfortunately, computing the eigenvalues and eigenvectors of a matrix is a difficult numerical problem. This so-called eigenproblem can be solved by a number of iterative heuristics which require $\Omega(p^3)$ operations to compute all the eigenpairs of a p−by−p matrix [15]. If $p > n$, then the computational cost may be reduced by computing the eigendecomposition of $A^T A$, then using the fact if (v_i, α_i) is an eigenvector-eigenvalue pair for A^TA, then $A(A^TAv_i) = A(\alpha_i v_i)$, hence (Av_i, α_i) is an eigenvector-eigenvalue pair for AA^T. The algorithm can then easily compute the r-most significant eigenvectors of AA^T from the eigenvectors A^TA for a total cost of $O(prn)$ operations. Nevertheless, the eigendecomposition cost of $\Omega(n^3)$ operations is still substantial, suggesting that PCA should not be used as an NMF initialization technique if both p and n are large.

5.3.3 NNDSVD

The next low rank approximation initialization procedure we discuss is the Nonnegative Double Singular Value Decomposition (NNDSVD) due to Boutsidis and Gallopoulos in 2008 [16]. As its name suggests, this method relies on two singular value decompositions (SVDs). The SVD of a rank-k matrix $X \in \mathbb{R}^{p\times n}$ decomposes X into $X = U\Sigma V^T$, where $U \in \mathbb{R}^{p\times k}$ and is orthogonal, $V \in \mathbb{R}^{n\times k}$ and is orthogonal, and $\Sigma \in \mathbb{R}^{k\times k}$ and is diagonal. The columns of U are the eigenvalues of XX^T and form a basis for the column space of X, the columns of V are the eigenvalues of X^TX and form bases of the row and null spaces of X, and the diagonal entries of Σ are the singular values of X, i.e. the eigenvalues of X^TX. This decomposition can also be written as the sum of rank-1, so-called singular factors: $X = \sum_{i=1}^{r} \sigma_i u_i v_i^T$, where $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_r > 0$ are the r leading singular values of X, and $\{u_i, v_i\}_{i=1}^{r}$ are the corresponding columns of U and V, i.e. the left and right singular vectors. As in [16], let $C^{(j)} = u_j v_j^T$ from now on, and let the set $\{\sigma_j, u_j, v_j\}$ be referred to as the j-th singular triplet.

Importantly, the best rank-r approximation of X in the Frobenius norm, for all $r \leq k$, is given by the sum of the first r singular factors, namely

$$\arg \min_{\text{rank}(A)\leq r} \|X - A\|_F = \sum_{i=1}^{r} \sigma_i C^{(i)} \tag{5.23}$$

by the Eckart-Young theorem [16]. The right hand side is not necessarily nonnegative, so unfortunately we cannot decompose it into a NMF. Instead, the intuition for NNDSVD comes from computing a nonnegative approximation of the right hand side, which happens to have favorable mathematical properties.

To understand the algorithm in detail we must introduce additional notation. For any matrix or vector Y, let $Y_+ = \max(Y, 0)$ be the positive section of Y and $Y_- = \max(-Y, 0)$ be the negative section, where max and $|\cdot|$ are again element-wise operations.

The NNNDSVD algorithm first computes the leading r singular triplets of X, then for each of the $\{C^{(j)}\}_{j=1}^{r}$ matrices associated with the singular triplets, computes the maximum singular triplet of the positive section $C_{+}^{(j)}$, and uses these triplets to initialize W and H [16]. The authors show that for all j, $C_{+}^{(j)}$ has rank at most 2, and if $\{\sigma_i, u_i, v_i\}$ is the singular triplet that yields $C^{(j)}$, then the two singular triplets which compose $C_{+}^{(j)}$ are $\{\|u_{j,+}\|\|v_{j,+}\|, \frac{u_{j,+}}{\|u_{j,+}\|}, \frac{v_{j,+}}{\|v_{j,+}\|}\}$ and $\{\|u_{j,-}\|\|v_{j,-}\|, \frac{u_{j,-}}{\|u_{j,-}\|}, \frac{v_{j,-}}{\|v_{j,-}\|}\}$, where $u_{j,+}$ is the positive section of the j^{th} column of U. Note that both of these singular triplets contain only nonnegative elements. If $\|u_{j,+}\|\|v_{j,+}\| > \|u_{j,-}\|\|v_{j,-}\|$, the algorithm sets the j-th column of $W^{(0)}$ and the j-th row of $H^{(0)}$ to reconstruct the 'positive' singular triplet as follows [16]:

$$w_{:,j}^{(0)} = \sqrt{\sigma_j \|u_{j,+}\|\|v_{j,+}\|}\frac{u_{j,+}}{\|u_{j,+}\|}, \quad h_{j,:}^{(0)} = \sqrt{\sigma_j \|u_{j,+}\|\|v_{j,+}\|}\frac{v_{j,+}^{T}}{\|v_{j,+}\|} \tag{5.24}$$

Otherwise, it sets them to reconstruct the 'negative' singular triplet:

$$w_{:,j}^{(0)} = \sqrt{\sigma_j \|u_{j,-}\|\|v_{j,-}\|}\frac{u_{j,-}}{\|u_{j,-}\|}, \quad h_{j,:}^{(0)} = \sqrt{\sigma_j \|u_{j,-}\|\|v_{j,-}\|}\frac{v_{j,-}^{T}}{\|v_{j,-}\|} \tag{5.25}$$

This procedure holds for all $j = 2, 3, ..., r$. Since X is nonnegative, its leading singular triplet is also homogeneous in sign, so its element-wise absolute value can used to initialize the first column of W and the first row of H directly [16]. We give a full overview of the NNDSVD algorithm in Algorithm 8.

```
Algorithm 8: NNDSVD [16]
  Input: Nonnegative matrix $X \in \mathbb{R}^{p \times n}$, integer $r \leq \min(p, n)$
  Output: Nonnegative matrices $W^{(0)} \in \mathbb{R}^{p \times r}$ and $H^{(0)} \in \mathbb{R}^{r \times n}$, where $r = k$
    Compute the $r$ largest singular factors of $X$: $\{\sigma_i, u_i, v_i\}_{i=1}^{r}$
    Set $w_{:,1}^{(0)} = |\sqrt{\sigma_1} u_1|$ and $h_{1,:}^{(0)} = |\sqrt{\sigma_1} v_1^T|$
    for $j = 2, 3, ..., r$ do
        Let $x = u_{:,j}$ and $y = v_{:,j}$
        if $\|x_+\|\|y_+\| > \|x_-\|\|y_-\|$ then
            Set $w_{:,j}^{(0)} = \sqrt{\sigma_j \|x_+\|\|y_+\|} \frac{x_+}{\|x_+\|}$
            Set $h_{j,:}^{(0)} = \sqrt{\sigma_j \|x_+\|\|y_+\|} \frac{y_+^T}{\|y_+\|}$
        else
            Set $w_{:,j}^{(0)} = \sqrt{\sigma_j \|x_-\|\|y_-\|} \frac{x_-}{\|x_-\|}$
            Set $h_{j,:}^{(0)} = \sqrt{\sigma_j \|x_-\|\|y_-\|} \frac{y_-^T}{\|y_-\|}$
        end if
    end for
```

We next discuss the mathematical justification that Boutsidis and Gallopoulos give for why this technique is effective. First, they prove the following lemma:

Lemma 1. *[16] Consider any rank-1 matrix $C \in \mathbb{R}^{p \times n}$. Then rank($C_+$), rank($C_-$) ≤ 2.*

Proof. Since C is rank 1, we can write

$$C = xy^T \tag{5.26}$$

$$= (x_+ - x_-)(y_+ - y_-)^T \tag{5.27}$$

$$= (x_+ y_+^T + x_- y_-^T) - (x_+ y_-^T + x_- y_+^T) \tag{5.28}$$

$$\tag{5.29}$$

The positive and negative sections of both x and y have nonzero entries at complementary entries within x and y, so the four terms in the last expression on the right-hand side have complementary nonzero entries as a foursome (meaning for each index (i, j), only one of the four terms may have a nonzero entry at that index). Furthermore, each of the four terms are nonnegative, so it follows that $C_+ = x_+y_+^T + x_-y_-^T$ and $C_- = x_+y_-^T + x_-y_+^T$. Thus, both C_+ and C_- have ranks of at most 2 [16]. $\square$

This result helps us to construct the SVD of the positive section $C_+^{(j)}$ for each j. In fact, by slightly rewriting our equation for $C_+^{(j)}$ above, we will have written $C_+^{(j)}$ as the sum of its singular factors. First recall that here, $C^{(j)} = u_j v_j^T$, so $x \equiv u_j$ and $y \equiv v_j$. Then the previous equation implies [16]:

$$C_+^{(j)} = u_{j,+}v_{j,+}^T + u_{j,-}v_{j,-}^T \tag{5.30}$$

$$= \|u_{j,+}\|\|v_{j,+}\| \frac{u_{j,+}}{\|u_{j,+}\|} \frac{v_{j,+}^T}{\|v_{j,+}\|} + \|u_{j,-}\|\|v_{j,-}\| \frac{u_{j,-}}{\|u_{j,-}\|} \frac{v_{j,-}^T}{\|v_{j,-}\|} \tag{5.31}$$

Note that $\|u_{j,+}\|\|v_{j,+}\|$ and $\|u_{j,-}\|\|v_{j,-}\|$ are nonnegative, and since the vectors $u_{j,+}$ and $u_{j,-}$ have complementary nonzero entries, they are orthogonal (and the same holds for $v_{j,+}$ and $v_{j,-}$). Thus, $\sigma^{(j,+)} = \text{diag}([\|u_{j,+}\|\|v_{j,+}\|, \|u_{j,-}\|\|v_{j,-}\|])$, $U^{(j,+)} = [u_{j,+}, u_{j,-}]$, and $V^{(j,+)} = [v_{j,+}, v_{j,-}]$, satisfy the conditions of the SVD, so they must be the SVD by uniqueness of the SVD [16]. Next, to best approximate $C^{(j)}$, it makes sense to take the larger singular factor of $C_+^{(j)}$ (in terms of its singular value), and we know that each of the entries of the singular vectors must be nonnegative, confirming that the algorithm is feasible.

Boutsidis and Gallopoulos provide an error bound that establishes the minimal effectiveness of this algorithm. Their reasoning proceeds as follows [16].

First, recall that $\{\sigma_j, u_j, v_j\}_{j=1}^k$ denotes the full set of non-trivial singular triplets of the rank-k matrix X, and they are sorted in decreasing order of σ_j. Similarly, let $\{\sigma_i(C_+^{(j)}), x_i(C_+^{(j)}), y_i(C_+^{(j)})\}_{i=1}^2$ denote the full set of non-trivial singular triplets of $C_+^{(i)}$, ordered such that $\sigma_1(C_+^{(j)})$ is the larger of the two singular values [16].

Next, let $X^{(r)}$ denote the best rank-r approximation of X in the Frobenius norm, and $E^{(r)} = A - A^{(r)}$ denote the resulting residual. Then,

$$E^{(r)} = \sum_{j=r+1}^{k} \sigma_j u_j v_j^T \tag{5.32}$$

$$X^{(r)} = \sum_{j=1}^{r} \sigma_j u_j v_j^T \tag{5.33}$$

$$= \sigma_1 C^{(1)} + \sum_{j=2}^{r} \sigma_j C_+^{(j)} - \sum_{j=2}^{r} \sigma_j C_-^{(j)} \tag{5.34}$$

$$= \sigma_1 C^{(1)} + \sum_{j=2}^{r} \sigma_j \sigma_1(C_+^{(j)}) x_1(C_+^{(j)}) y_1(C_+^{(j)})^T + \hat{E} \tag{5.35}$$

where

$$\hat{E} \coloneqq \sum_{j=2}^{k} \sigma_j \sigma_2(C_+^{(j)}) x_2(C_+^{(j)}) y_2(C_+^{(j)})^T - \sum_{j=2}^{r} \sigma_j C_-^{(j)} \tag{5.36}$$

By the construction of the NNDSVD algorithm, we have that (for simplicity we drop the (0) superscript in the notation for the initial estimates of W and H)

$$WH = \sigma_1 C^{(1)} + \sum_{j=2}^{r} \sigma_j \sigma_1(C_+^{(j)}) x_1(C_+^{(j)}) y_1(C_+^{(j)})^T = X^{(r)} - \hat{E} \tag{5.37}$$

This implies that $X - WH = X^{(r)} + E^{(r)} - (X^{(r)} - \hat{E}) = E^{(r)} + \hat{E}$. Thus,

$$\|E^{(r)}\|_F \leq \|X - WH\|_F \leq \|E^{(r)}\|_F + \|\hat{E}\|_F \tag{5.38}$$

where the last inequality follows from the Triangle Inequality [16]. Thus, the Frobenius norm of the error in the initial rank-r estimate WH of X is larger than the smallest possible error from a rank-r estimate by at most $\|\hat{E}\|_F$. In order to better understand this bound, we can in turn bound $\|\hat{E}\|_F$. To do so, we first must bound $\|x_1(C_+^{(j)}) y_1(C_+^{(j)})^T\|_F$ and $\|C_-^{(j)}\|_F$. First consider the former term, and rewrite

$a = x_1(C_+^{(j)})$ and $b = y_1(C_+^{(j)})$ [16].

$$
\begin{aligned}
\|ab^T\|_F &= \Big(\sum_{i=1}^{p}\sum_{l=1}^{n}(a_i b_l)^2\Big)^{1/2} && \tag{5.39}\\
&= \Big(\sum_{i=1}^{p}(a_i^2 \sum_{l=1}^{2} b_l^2)\Big)^{1/2} && \tag{5.40}\\
&= \Big(\sum_{i=1}^{p} a_i^2\Big)^{1/2} \qquad \text{because } \|b\|_2^2 = 1 && \tag{5.41}\\
&= 1 \qquad \text{because } \|a\|_2^2 = 1 && \tag{5.42}\\
& && \tag{5.43}
\end{aligned}
$$

Thus, $\|x_1(C_+^{(j)})y_1(C_+^{(j)})^T\|_F = 1$. A similar proof shows that $\|C^{(j)}\|_F = \|u_j v_j^T\|_F = 1$. Since $C_+^{(j)}$ and $C_-^{(j)}$ have complementary nonzero entries, $\|C_+^{(j)}\|_F^2 + \|C_-^{(j)}\|_F^2 = \|C^{(j)}\|_F^2 = 1$, thus both $\|C_\pm^{(j)}\|_F \leq 1$. Now, by the Triangle Inequality, we have that

$$
\begin{aligned}
\|\hat{E}\|_F &\leq \sum_{j=2}^{r} \|\sigma_j \sigma_2(C_+^{(j)}) x_2(C_+^{(j)}) y_2(C_+^{(j)})^T - \sigma_j C_-^{(j)}\|_F && \tag{5.44}\\
&= \sum_{j=2}^{r} \big(\|\sigma_j \sigma_2(C_+^{(j)}) x_2(C_+^{(j)}) y_2(C_+^{(j)})^T\|_F + \|\sigma_j C_-^{(j)}\|_F\big) \quad \text{(the nonzero entries do not overlap)} && \tag{5.45}\\
&\leq \sum_{j=2}^{r} (\sigma_2(C_+^{(j)}) + 1)\sigma_j && \tag{5.46}
\end{aligned}
$$

The final inequality is our improvement on the upper bound of $2\sum_{j=2}^{r}\sigma_j$ given in [16], and follows by recalling that the Frobenius norm of a matrix is equal to the sum of its singular values, so because the Frobenius norm of $C_+^{(j)}$ is less than or equal to 1, and $\sigma_2(C_+^{(j)})$ is its second-largest singular value, it can be at most $\frac{1}{2}$. This is still a loose upper bound, as we can easily see that it may be larger than the error from trivially initializing W and H with all zero entries. Thus we judge the effectiveness of the algorithm by how it performs in practice, measured by how quickly it enables NMF algorithms to converge and the error at convergence, instead of celebrating it because of this loose upper bound on its accuracy. Nevertheless, it is nice to have even a loose upper bound to have some validation of our intuitions, especially for solving a problem which has traditionally precluded theoretical guarantees.

Meanwhile, it is important to note that the NNDSVD algorithm computes W and H that are roughly 50% sparse because the singular vectors u_j, v_j have roughly half positive and half negative elements for all $j > 1$ (also note that u_j and v_j are guaranteed to have at least one positive and one negative entry, because they are orthogonal to u_1 and v_1, respectively, which are homogeneous in sign, thus W and H are guaranteed to have no columns or rows that are all zero, respectively [69]). The high sparsity of (W, H) is desirable if X is sparse but becomes problematic when X is dense. To satisfy the dense case, Boutsidis and Gallopoulos introduce two variants of NNDSVD which perturb the zero entries in $W^{(0)}$ and $H^{(0)}$ in varying ways [16]. Our implementation assumes nothing about the sparsity of X, so we employ a flexible method of the same spirit which perturbs zero elements of $W^{(0)}$ and $H^{(0)}$ until those matrices reach a level of sparsity commensurate with the sparsity of X. Namely, if X has fewer zero entries than $W^{(0)}$ or $H^{(0)}$, we randomly perturb at most ten of the zero entries in each of $W^{(0)}$ and $H^{(0)}$ until $\|X\|_0 \leq \|W^{(0)}\|_0$ and $\|X\|_0 \leq H^{(0)}\|_0$. We perturb them according to the same procedure as one of the NNDSVD variants, by assigning a randomly sampled value from the interval $[0, \mu(X)/100]$ to the perturbed entry, where $\mu(X)$ denotes the mean of X.

Finally, the computational cost of NNDSVD depends primarily on the cost to compute the largest r singular triplets of X and to compute the largest singular triplet of the positive section of each of the $r-1$ of the singular factors of X. Iterative algorithms exist to solve the first computational task, and we can assume they run in $O(prn)$ operations [15], although this assumption treats the number of iterations as a constant, despite the fact that it may be large. Fortunately, the second computational task is less expensive, since there are only two non-trivial singular triplets of each of the positive sections of the singular factors of X, and they can be computed easily once we have the singular triplets of X. Indeed, computing them requires only separating the positive and negative elements of p- and n-dimensional vectors, and taking the norms of the new vectors, which can be done in $O(p + n)$ operations for each of the $r - 1$ singular triplets (there is no need to compute the matrices $C^{(j)}$). This makes for a cost of $O(r(p + n))$ operations for all of the $r - 1$ singular triplets, thus the total cost of the algorithm is $O(prn)$, which is the same complexity as the clustering algorithms and the NMF heuristics.

5.3.4 SVD-NMF

Perhaps the major shortcoming of the NNDSVD algorithm is that it discards the negative sections of the singular factors $\{C^{(j)}\}_{j=2}^{r}$ as well as half of the singular factors of the positive sections, so great deal of information about X is lost in the approximation. Qiao observed that abbreviating the NNDSVD algorithm by taking the absolute value of the full singular factors $\{C^{(j)}\}_{j=2}^{r}$, then using these modified singular factors and their associated singular values to initialize W and H often performed better than the NNDSVD method in practice [66]. He called this algorithm the SVD-NMF algorithm, which we explain in detail in 9. Note that $|A|$ is the element-wise absolute value operation on A.

Algorithm 9: SVD-NMF [66]

```
**Input:** Nonnegative matrix $X \in \mathbb{R}^{p\times n}$, integer $r \leq \min(p, n)$
**Output:** Nonnegative matrices $W^{(0)} \in \mathbb{R}^{p\times r}$ and $H^{(0)} \in \mathbb{R}^{r\times n}$
  Compute the $r$ largest singular factors of $X$: $\{\sigma_i, u_i, v_i\}_{i=1}^{r}$
  Set $w_{:,1}^{(0)} = |u_1|$ and $h_{1,:}^{(0)} = |\sigma_1 v_1^T|$
  **for** $j = 2, 3, ..., r$ **do**
      Set $w_{:,j}^{(0)} = |u_j|$ and $h_{j,:}^{(0)} = \sigma_j |v_j|$
  **end for**
```

SVD-NMF has the same computational complexity ($O(prn)$) as NNDSVD due to the cost of computing the r leading singular triplets of X. The cost of computing the absolute values of the singular vectors is order-wise equivalent to the cost of the post-processing of the same vectors in the NNDSVD algorithm. We will investigate Qiao's claim that the SVD-NMF algorithm yields a more effective NMF initialization for this same cost [66] through our experimental testing in Chapter 6. One notable characteristic of the SVD-NMF algorithm is that it generates dense W and H because the singular factors of a matrix are dense, regardless of the sparsity of the matrix they factor. This may pose issues if X is sparse, which we will test.

5.3.5 NNSVD-LRC

One drawback of both the NNDSVD and SVD-NMF algorithms is that their approximation errors do not necessarily decrease as the rank of the factorization increases. To see this, let $X^{(r)} = W^{(0)}H^{(0)}$ be the rank-r nonnegative approximation of X yielded by either the NNDSVD or the SVD-NMF algorithms. Because each algorithm computes X_r such that it is the sum of r nonnegative matrices, we must have that $X^{(r+1)} \geq X^{(r)}$ (each element of $X^{(r+1)}$ is greater than the corresponding element in $X^{(r)}$. Therefore, there may be some r' such that $\|X - X^{(r')}\|_F$ increases for every $r \geq r'$ [69]. However, we would like our approximation to improve as we are allowed more degrees of freedom; for the approximation quality to diminish with more degrees of freedom is counterproductive and wasteful.

To fix this issue, Syed et al. recently proposed an alternative SVD initialization which tries to reduce the amount of information lost when modifying the SVD to satisfy the nonnegativity constraints [69]. Their algorithm uses a few iterations of low rank NMF after initializing with an SVD-based method, hence they call their algorithm nonnegative singular value decompostion with low rank correction (NNDSVD-LRC).

To explain the algorithm, we use the same notation that we used in the NNDSVD section to denote the positive and negative sections of matrices and vectors. Namely, let $A_+ = \max(A, 0)$ and $A_- = \max(-A, 0)$, where A is a matrix or vector and max is an element-wise absolute value operation. Furthermore, if $\{\sigma_j, u_j, v_j\}_{j=1}^k$ are the leading k singular triplets of X, let $C^{(j)} = u_j v_j^T$ be the j-th singular factor of X. Recall that

$$C^{(j)} = C_+^{(j)} - C_-^{(j)} \tag{5.47}$$

where

$$C_+^{(j)} = u_{j,+}v_{j,+} + u_{j-}v_{j,-} \tag{5.48}$$

$$C_-^{(j)} = u_{j,+}v_{j,-} + u_{j-}v_{j,+} \tag{5.49}$$

and each of the terms above are singular factors of the matrix on their respective left hand side. Instead of discarding one of the singular factors of $C_+^{(j)}$, as NNDSVD does, or obscuring $C_+^{(j)}$ by adding $C_-^{(j)}$ to it, as SVD-NFM does, NNSVD-LRC incorporates both the singular factors of $C_+^{(j)}$ into its initial estimate of (W, H). As such, each $C_+^{(j)}$, besides $C_+^{(1)}$, determines the initial estimate of two columns of W and two rows of H. In particular, for all $j = 2, 3, 4, ...$, the algorithm initializes $w_{:,j}$ and $h_{j,:}$ as follows until all columns and rows have been initialized [69]:

$$w_{:,j} = \begin{cases} \sqrt{\sigma_{(j/2)+1}}u_{(j/2)+1,+}, & \text{if } j \text{ is even} \\ \sqrt{\sigma_{\lfloor(j/2)+1\rfloor}}u_{\lfloor(j/2)+1\rfloor,+}, & \text{otherwise} \end{cases} \tag{5.50}$$

$$h_{j,:} = \begin{cases} \sqrt{\sigma_{(j/2)+1}} v^T_{(j/2)+1,+}, & \text{if } j \text{ is even} \\ \sqrt{\sigma_{\lfloor (j/2)+1 \rfloor}} v^T_{\lfloor (j/2)+1 \rfloor,+}, & \text{otherwise} \end{cases} \tag{5.51}$$

Thus, we only need to compute the leading $k := \lceil \frac{r}{2} + 1 \rceil$ singular triplets of X to initialize all the columns of W and all the rows of H. Like the NNDSVD and SVD-NMF algorithms, the NNSVD-LRC algorithm initializes $w_{:,1}$ and $h_{1,:}$ with the absolute value of the factors of $C^{(1)}$, since these factors must be either nonnegative or nonpositive by Perron-Frobenius theory [69]. Additionally, similarly to the NNDSVD algorithm, the NNSVD-LRC algorithm yields initial W and H that are roughly 50% sparse because $u_{j,+}, u_{j,-}, v_{j,+}$ and $v_{j,-}$ are roughly 50% sparse for all $j > 1$. Intuition suggests this is useful for factoring sparse matrices but potentially problematic for factoring dense matrices. However, the second stage of the algorithm reduces this sparsity.

Unlike the NNDSVD and SVD-NMF algorithms, the algorithm has a second phase which does not involve computing another SVD. In this so-called 'low rank correction' phase, the algorithm runs a few iterations of a NMF algorithm to solve the problem instance

$$\min_{W \in \mathbb{R}^{p \times r}_{\geq 0}, H \in \mathbb{R}^{r \times n}_{\geq 0}} \tfrac{1}{2} \|X_k - WH\|_F^2 \tag{5.52}$$

where X_k is the rank-k SVD approximation of X formed by the sum of the leading k singular factors. Syed et al. use the accelerated HALS heuristic to execute the NMF iterations in their implementation [69], and execute one iteration at a time until the change in error on two iterations is less than some fraction δ of the initial error $e_0 = \|X_k - W^{(0)} H^{(0)}\|_F^2$, where $W^{(0)}$ and $H^{(0)}$ are the matrices generated from the first stage of the algorithm. Indeed, the algorithm uses $W^{(0)}$ and $H^{(0)}$ to initialize the first call to the NMF algorithm, and on subsequent calls uses the current estimates of W and H as the initializations and updates them by setting them equal to the matrices returned by the algorithm. Syed et al. set $\delta = 0.05$, and observe that they only need to execute less than 10 iterations to satisfy the stopping condition for each of the datasets they test. For consistency with our tests of NMF heuristics, we use the non-accelerated version of HALS in our implementation, and use the same stopping condition employed by Syed et al, with a maximum of 50 calls to HALS allowed [69]. A detailed overview of NNDSVD-LRC is shown in Algorithm 10.

Algorithm 10: NNDSVD-LRC [69]

```
Input: Nonnegative matrix $X \in \mathbb{R}^{p \times n}$, positive integer $r < \min(p, n)$
Output: Nonnegative matrices $W \in \mathbb{R}^{p \times r}$ and $H \in \mathbb{R}^{r \times n}$
  Let $k := \lceil \frac{r}{2} + 1 \rceil$
  Compute the truncated rank-$k$ SVD of $X$: $\{\sigma_i, u_i, v_i\}_{i=1}^{k}$
  Set $w_{:,1} = |\sqrt{\sigma_1} u_1|$ and $h_{1,:} = |\sqrt{\sigma_1} v_1^T|$
  Set $l = 2$
  for $j = 2, 3, ..., p$ do
      if $j$ is even then
          Set $w_{:,j} = \sqrt{\sigma_l} u_{l,+}$
          Set $h_{j,:} = \sqrt{\sigma_l} v_{l,+}^T$
      else
          Set $w_{:,j} = \sqrt{\sigma_j} u_{l,-}$
          Set $h_{j,:} = \sqrt{\sigma_j} v_{l,-}^T$
          Increment $l = l + 1$
      end if
  end for
  Set $e_0 = \|X_{(k)} - WH\|_F$, where $X_{(k)} = U\Sigma V^T$ is the rank-$k$ SVD approximation
  of $X$
  Initialize $q = 0$
  while $q = 0$ or $e_q - e_{q-1} > \delta e_0$ do
      Perform one iteration of HALS with target matrix $X_{(k)}$ and initial matrices
      $(W, H)$; update $(W, H)$ with the output
      Compute $e_{q+1} = \|X_{(k)} - WH\|_F$
      Increment $q = q + 1$
  end while
```

It may seem like cheating to use an NMF algorithm here. After all, the purpose of NNSVD-LRC is to efficiently compute an effective initialization for NMF algorithms; why should using one of those NMF algorithms to find a better NMF be considered part of the initialization process? The answer stems from the fact that we are running the NMF algorithm on a low-rank approximation of X, not X itself. As Syed et al. note, most of the cost of running NMF algorithms comes from computing XH^T, W^TX, HH^T and W^TW [69]. The latter two matrices can be computed in $O(r^2n)$ and $O(r^2p)$ operations, respectively, but the first two each require $O(prn)$ operations in general. Fortunately, since in our case $X_{(k)}$ has rank k and its rank-k factorization is known, the first two matrices can each be computed in $O(r^2(p+n))$ operations (as $k = O(r)$). Thus, the total cost of each NMF iteration is only $O(r^2(p+n))$ operations, as opposed to $O(prn)$. Syed et al. also show that the error e_{q+1} can be computed in $O(r^2(p+n))$ operations on each iteration by taking advantage of the low rank of $X_{(k)}$ [69]. Since we are usually executing a very small number of iterations [69], this low rank correction phase is much cheaper than executing typical NMF iterations, and therefore can be distinguished from typical NMF iterations.

Importantly, Syed et al. claim there is a significant benefit to this low rank correction phase that comes with its low cost [69]. Since $X_{(k)}$ contains both $C_+^{(j)}$ and $C_-^{(j)}$ for all $j \leq k$, the low rank correction phase allows the estimates for W and H to account for the negative sections of the leading k singular factors of X that were previously discarded. In general, using a low rank approximation of X as the target matrix is an intriguing approach to solve NMF problems, and we explore the low rank approximation-based NMF algorithm due to Cichocki et al. [84] in Section 7.1.2.

The computational complexity of the NNDSVD phase of the NNSVD-LRC algorithm is $O(prn)$ to compute the truncated SVD of X, but this cost has roughly half the constant associated with it compared to the NNDSVD and SVD-NMF algorithms because it only has to compute $\lceil r/2 + 1 \rceil$ singular triplets as opposed to r singular triplets. Combine this with the $O(r^2(p+n))$ cost from the low-rank correction phase, and the total cost is $O(prn)$ because r is bounded above by $\min(p, n)$. After empirical investigations, Syed et al. claim that not only is the NNSVD-LRC less expensive than the other SVD algorithms, but it finds a better initialization than they do (in terms of leading to faster NMF convergence, not necessarily in reducing the final error) [69]. We will investigate how well the efficacy of this claim in our own empirical tests in the next chapter.

Chapter 6

Tests of Initialization Techniques

In this section we test how well the initialization techniques discussed in the previous section improve NMF algorithm performance on both synthetic and real datasets. These empirical tests are very insightful in regards to the effectiveness of the initialization techniques because theoretical analysis of the techniques has proven difficult. Naturally, we evaluate NMF algorithm performance with a certain initialization in terms of its convergence rate and final error. We execute each test by computing a rank-r NMF of a different type of data stored in the matrix $X \in \mathbb{R}^{p \times n}$, and employ the default setting of parameters:

$$p = 50, \quad n = 250, \quad r = 10 \tag{6.1}$$

For each test on a particular type of data, we run four NMF algorithms (multiplicative updates (MU), hierarchical alternating least squares (HALS), alternating nonnegative least squares with block principle pivoting (ANLS BPP), and projected gradient descent (PGD)) with each of the twelve initialization methods we discussed in the previous section (including the random initialization) and plot the relative error yielded by the algorithms over time for each initialization.

We do not include the time taken to compute the initialization in the error curves we plot. This is because some methods, such as subtractive clustering, NNDSVD, and SVD-NMF, take significantly longer than the other initializations to execute, which makes comparing the relative error curves they generate difficult if some are shifted a ways horizontally. Nevertheless, to account for the speed of each initialization computation, we share the average execution time for each technique across each test in this chapter on 50-by-250 matrices in Table 6.1. The execution times are not

	Random	\textit{Xcol}	Rand-c	Co-occurrrence	*k*-means	S. *k*-means	Fuzzy C-means	Sub. Clustering	PCA	NNDSVD	NNDSVD-LRC	SVD-NMF
Time (ms)	0.183	0.842	0.925	0.616	11.2	9.02	4.65	99.2	1.27	72.2	4.95	59.2

Table 6.1: Execution times for initialization methods for NMF with $p = 50$, $n = 250$ and $r = 10$.

surprising, with the data column subset selection methods executing the fastest and the low rank approximation and clustering techniques executing substantially slower.

The first relative error plotted is the relative error of the initial factor estimates returned by the initialization method. As before, we define the relative error of a particular estimate (W, H) of an NMF of X as

$$\frac{\|X - WH\|_F}{\|X\|_F} \tag{6.2}$$

We do not test how the initialization techniques affect all of the algorithms that require initialization in the interest of space. Instead, we choose to only test HALS and ANLS BPP because of their superior performance in Chapter 3, and MU and PGD because they are very different types of algorithms and may shed further light on the behavior of the initialization methods. The plotted results from each test are the averages over 40 trials. We begin by testing on similar types of synthetic data as we did in Section 3.1.

6.1 Tests on Synthetic Data

6.1.1 Vanilla Gaussian Data

Our first tests of the initialization techniques are on matrices X in which each element is the absolute value of an independent random sample from the standard normal distribution. Figure 6.1 shows the NMF objective function value over time yielded by the four NMF algorithms for each of 12 different initialization strategies.

A few aspects of the results stand out as particularly interesting, and are signs of potential trends to look out for in future tests:

- NMF algorithm performance can be highly dependent on the initialization technique. We saw in our earlier tests in Chapter 3 that PGD tends to perform much worse than the rest of the NMF heuristics when initialized with the random initialization. But the results show that if PGD is initialized with the optimal methods for a particular type of data (here those methods are subtractive clus-

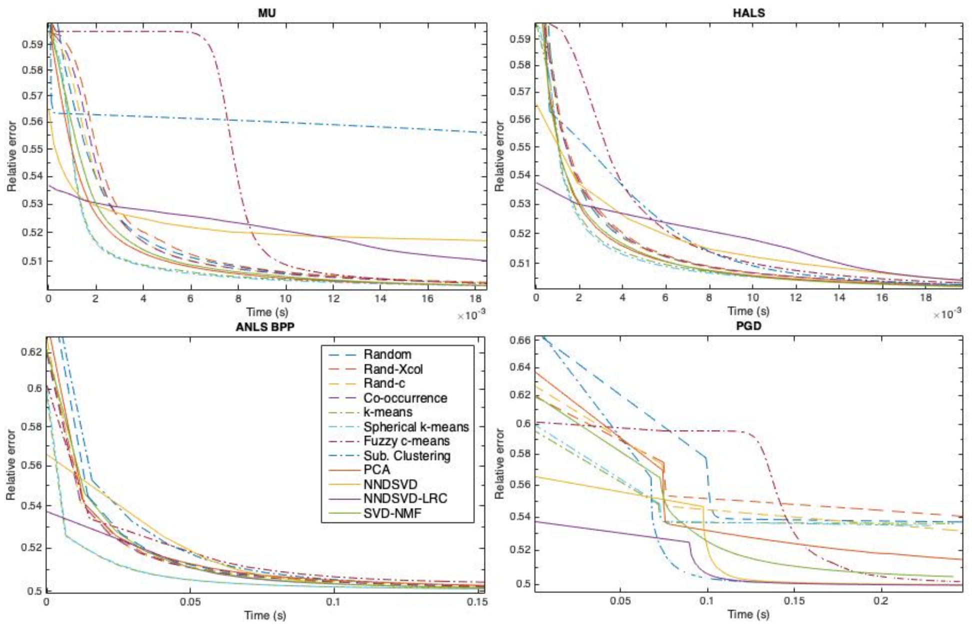


Figure 6.1: NMF relative error vs time for 12 different initialization strategies for (a) MU, (b) HALS, (c) ANLS BPP, and (d) PGD.

tering, NNDSVD, or NNDSVD-LRC), it can converge to a final relative error as small as ANLS BPP and in roughly the same amount of time. Likewise, MU, and to a lesser extent, HALS, see their performance on the data used here fluctuate based on the chosen initialization technique. ANLS BPP is the most robust to the initialization method, but even it demonstrates a range of convergence rates depending on the initialization technique.

- The random initialization performs average to below-average for initializing every algorithm. The random initialization can be seen as a baseline for initialization performance, so it is promising that many techniques improve on it.

- k-means and spherical k-means perform the best for initializing every algorithm besides PGD, and there is hardly any discrepancy between the error curves generated by these two algorithms. The predicted advantage of spherical k-means over k-means is likely not manifest because the columns of X are highly unlikely to be scalar multiples of each other, so there is no benefit of removing redundant information by normalizing them. Surprisingly, in nearly all cases these two clustering techniques perform substantially better than the other two clustering techniques, subtractive clustering and fuzzy c-means, suggesting that perhaps hard clustering is a more effective initialization strategy than soft clustering.

- The performance of NNDSVD and NNDSVD-LRC is especially intriguing. These two methods generate the least erroneous initial factors, but they yield error curves that tend to converge relatively slowly, especially for initializing MU. The sparsity of the factors that NNDSVD and NNDSVD-LRC yields may explain its poor performance for initializing this algorithm. MU has no way of making elements of the factors nonzero once they are already zero (see the update rules for MU in Section 2.1 to confirm this), so initializing MU with sparse factors limits the degrees of freedom the algorithm has to work with, and hence leads to a less accurate solution. The same may apply to HALS to a lesser extent, since each update is a closed form expression of X and the current factor estimates, so if the estimates are initially sparse then the products in are more likely to stay equal to zero or close to zero during the execution of the algorithm.

  NNDSVD and NNDSVD-LRC first compose W and H from the positive and/or negative components of singular vectors, on average 50% of the elements of those matrices are zero. Both methods have procedures that then attempt to make the factors less sparse, but the output initial estimates are still significantly sparse. NNDSVD perturbs zero elements in W and H until the *product* WH has sparsity commensurate with the sparsity of X, so W and H may each still remain sparse even when X is dense. In fact, we observe that in this test, NNDSVD outputs W and H that are 44.8% and 45.7% sparse on average, respectively. Meanwhile the few low-rank iterations of HALS at the end of the NNDSVD-LRC method do not eliminate the sparsity in the factors it generates, as the W yielded by NNDSVD-LRC had 25.9% zeros and the H had 21.1% zeros on average. None of the other NMF algorithms lose degrees of freedom in the same way as MU, and to a lesser extent HALS, when the initial factors are sparse, so it makes sense that these two initialization procedures perform the worst on MU and only slightly better on HALS. However, we are unsure why subtractive clustering also yields factors that converge so slowly for MU, since these initial factors are dense.

- The other SVD-based method, SVD-NMF, and the final low-rank approximation method, PCA, perform near the average for initializing all of the NMF algorithms besides HALS and MU, for which they are two of the best, behind

k-means and spherical k-means. Their success for initializing MU and HALS is likely explained at least in part by the fact that they generate dense W and H.

- The reason for the abnormally poor performance of subtractive clustering for initializing MU is unclear. Unlike the poor performance of NNDSVD and NNDSVD-LRC for MU, it cannot be because the initial factors W and H are sparse - subtractive clustering always initializes a dense H, and it initializes W with columns of X, so if X is dense, W will also be dense. Similarly, we doubt that the poor performance is an artifact of initializing W with columns of X, since other methods which do the same thing (random $Xcol$ and random-c) perform adequately for initializing MU. We may also rule out the hypothesis that subtractive clustering is a universally poor initialization technique, since it performs the best for initializing PGD. The other soft clustering technique, fuzzy c-means, also performs well for initializing only one algorithm (ANLS BPP).

- The data column subset selection methods (random *Xcol*, random-*c*, and co-occurrence) each perform roughly the same as the random initialization, which makes sense because there is not much structure behind their generation.

6.1.2 Sparsity

Multiple initialization strategies are claimed to be effective for factoring sparse data, so here we test how well the initializations perform on data with varying levels of sparsity, specifically datasets that are 50% and 95% sparse. We choose the indices of X with zero entries randomly, and generate the nonzero elements of X as usual by taking the absolute value of a random Gaussian sample. We display the results in Figures 6.2 and 6.3.

Assuming that a level of sparsity in the initial NMF factors similar to the level of sparsity in the data matrix conduces to better NMF algorithm performance, we expect that subtractive clustering and each of the column subset selection methods besides co-occurrence will perform relatively well on sparse data because the columns of their W matrices are also columns of X, so their W matrices have a commensurate level of sparsity with the sparsity of X. We also expect NNDSVD and NNDSVD-LRC to perform relatively well on sparse data because they yield W and H that are sparse when X is sparse [16]. On the other hand, the other methods generally yield dense factors even for sparse data, so we expect them to perform less well.

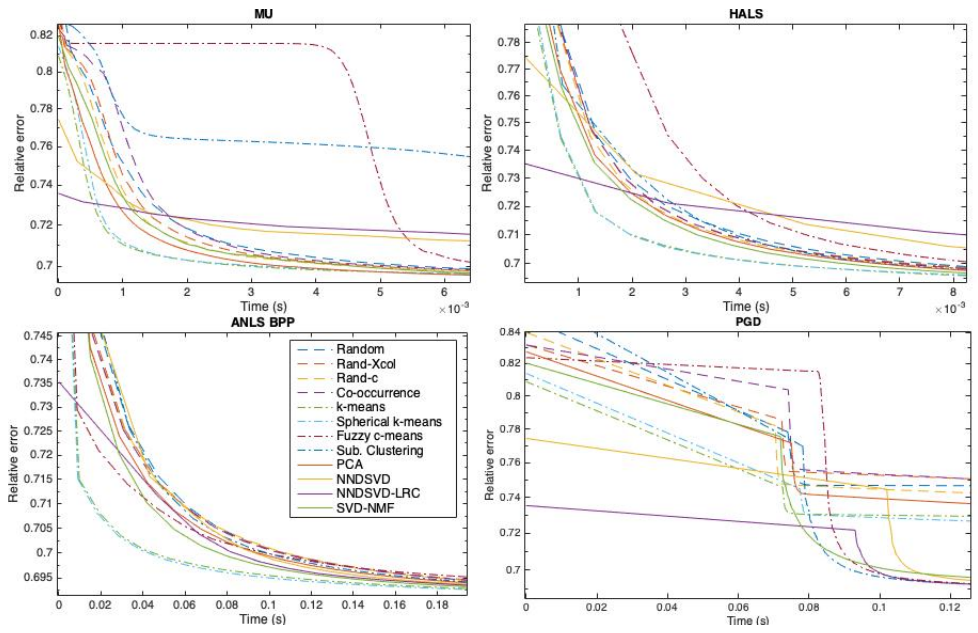


Figure 6.2: NMF relative error vs time for 12 different initialization strategies for (a) MU, (b) HALS, (c) ANLS BPP, and (d) PGD, with X 50% sparse.

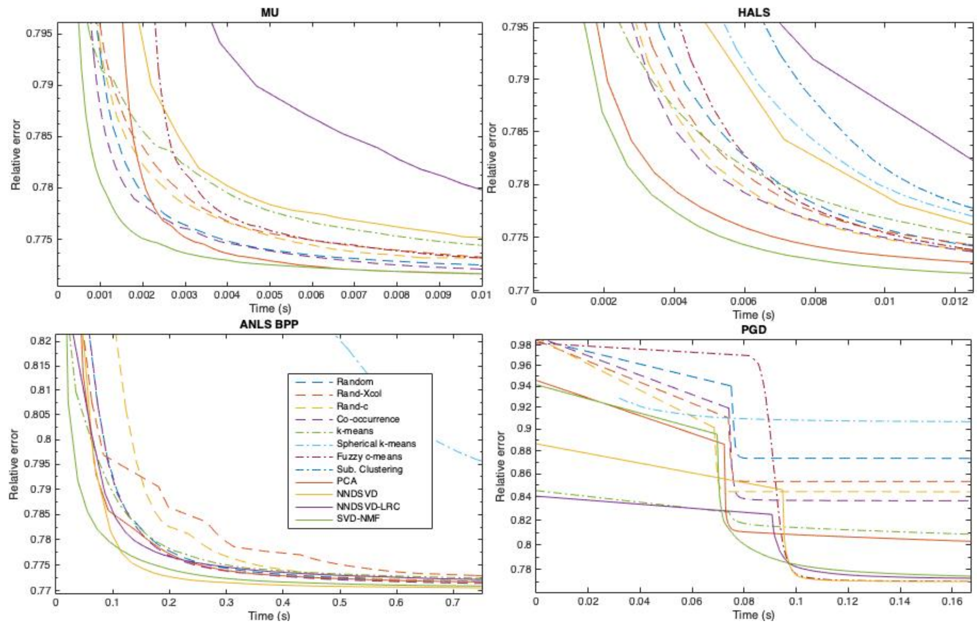


Figure 6.3: NMF relative error vs time for 12 different initialization strategies for (a) MU, (b) HALS, (c) ANLS BPP, and (d) PGD, with X 95% sparse.

However, the results when X is 50% and 95% sparse suggest that initial factors which are as sparse as the data matrix do not cause better NMF algorithm performance, especially for initializing MU and HALS. Table 6.2 supports this claim by showing the percentage of zeros in the initial factors yielded by notable initializa-

Table 6.2: Sparsity (percentage of zeros) in the factors yielded for certain types of data.

	Vanilla Gaussian data		Data 50% sparse		Data 95% sparse	
	W	H	W	H	W	H
k-means	0	0	0	0	62.4	45.4
Spherical k-means	0	0	0	0	41.2	27.4
Subtractive clustering	0	83.8	49.4	87.9	84.1	0
NNDSVD	44.8	45.7	46.8	46.2	42.0	47.6
NNDSVD-LRC	25.9	21.1	33.0	31.4	53.0	53.4

tion techniques for different types of data. These techniques are notable because the sparsity of their outputs as a function of the data is not obvious, and they exhibit substantial variation in performance across the different types of data. Across the table, larger sparsity in the factors corresponds to worse performance for initializing MU and HALS.

Another piece of evidence in support of this hypothesis that sparsity in the initial factors adversely affects MU and HALS algorithm performance is the the improved relative performance of PCA and SVD-NMF when X is 95% sparse. These initializations always yield dense factors, so it makes sense that when the other initializations yield increasingly sparse factors as X gets more and more sparse, their performance diminishes relative to the performance of SVD-NMF and PCA on MU and HALS. This applies to the five initialization techniques in Table 6.2, and also to the data column subset selection techniques, which by construction yield more sparse factors as X gets more sparse. It also makes sense that co-occurrence is arguably the most effective initialization method among the data column subset selection techniques when X is 95% sparse, since XX^T is likely less sparse than X in this scenario[3]. However, we are unsure why these trends are not also present in the results when X is 50% sparse.

Aside from these notes related to the sparsity of the initial factors, the only other notable change in the results when X is 50% and 95% sparse from the test in the previous section is the improved performance of fuzzy c-means and SVD-NMF for initializing PGD in both cases. When X is 95% sparse, we also note that NNDSVD exhibits a drastic improvement in performance, and k-means, fuzzy c-means, and

[3]If q is the probability that an element in X is nonzero, for all elements, then the probability that a particular element in XX^T is nonzero is nq^2, assuming no sum of nonzero products sums to 0. It follows that the probability that a particular element in XX^T is nonzero is greater than the probability that a particular element in X is nonzero if and only if $nq > 1$, which it is in this case.

subtractive clustering exhibit drastic decreases in performance, the reason for which we are unsure of.

6.1.3 Factorization Rank

Since many of the initialization techniques are low rank approximation methods, this begs the question of how such methods perform when their low-rank approximations of X may represent X exactly, in contrast with when they may not. An exact representation is only possible if the nonnegative rank of X is smaller than the factorization rank. Therefore, we test initialization performance when the factorization rank r is less than, equal to, and greater than the nonnegative rank of X in this section. Specifically, we hold the nonnegative rank of X constant at 10 and test $r = 5, 10$, and 15. To generate an X with nonnegative rank 10, we generate two matrices $U \in \mathbb{R}^{p\times 10}$ and $V \in \mathbb{R}^{10\times n}$ in which each element is the absolute value of an independent Gaussian sample, then set $X = UV$. We display the resulting error curves in Figures 6.4, 6.5, and 6.6.

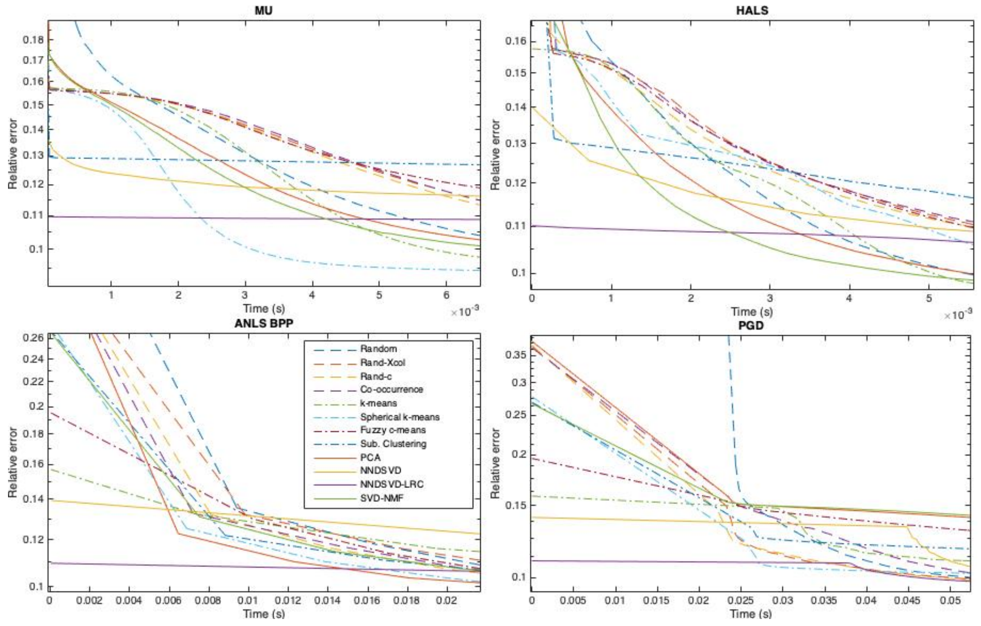


Figure 6.4: NMF relative error vs time for 12 different initialization strategies for (a) MU, (b) HALS, (c) ANLS BPP, and (d) PGD, with $r = 5 = \text{rank}_{\geq 0}(X)/2$.

As soon as the factorization rank is at least the nonnegative rank of the data, the NMF algorithms converge to a very accurate solution when initialized with most strategies, as expected. Other meta-trends are that the relative performance of initializations for MU and HALS remains mostly consistent across factorization rank,

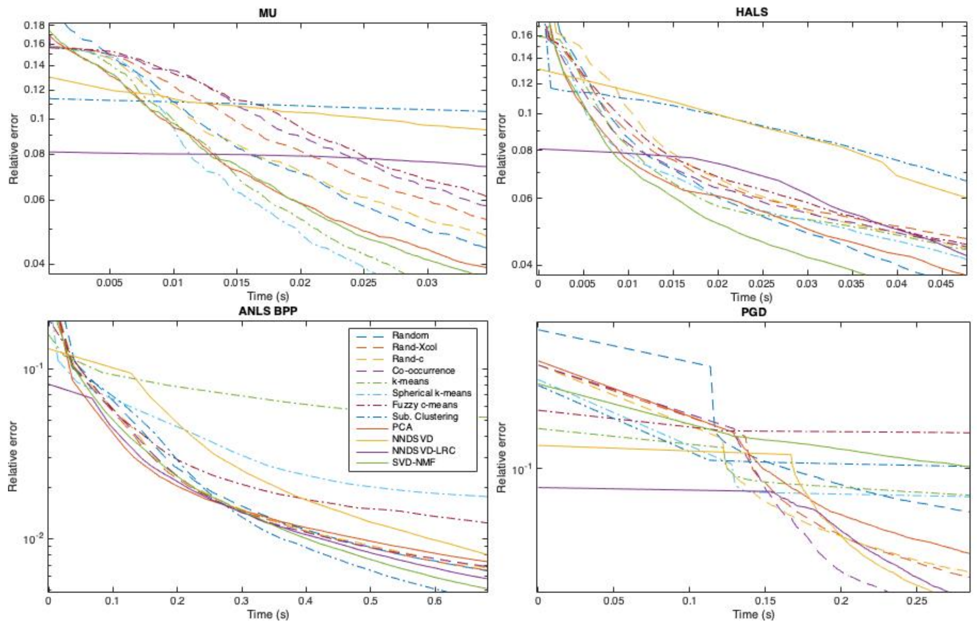


Figure 6.5: NMF relative error vs time for 12 different initialization strategies for (a) MU, (b) HALS, (c) ANLS BPP, and (d) PGD, with $r = 10 = \text{rank}_{\geq 0}(X)$.

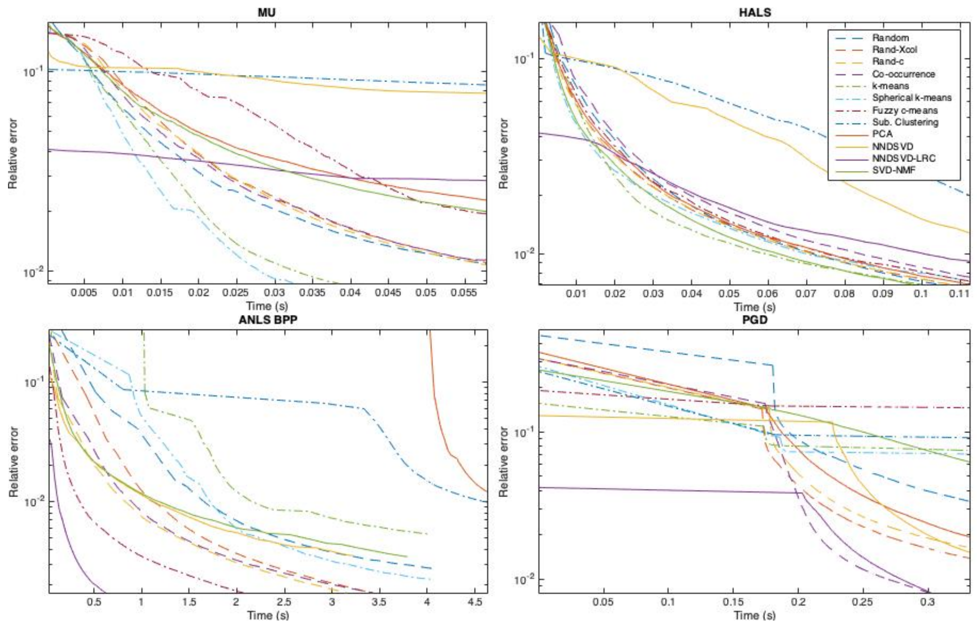


Figure 6.6: NMF relative error vs time for 12 different initialization strategies for (a) MU, (b) HALS, (c) ANLS BPP, and (d) PGD, with $r = 15 = 3\text{rank}_{\geq 0}(X)/2$.

with the exception being that the low rank approximation techniques tend to perform worse for initializing HALS as the factorization rank increases.

We expect to see the estimates of W and H yielded by NNDSVD and SVD-NMF have initial relative errors that cease to decrease with increasing factorization rank beyond some thresholds r' and r'' because WH is the sum of r nonnegative matrices, as we discussed in Section 5.3.5. In contrast, we expect the initial relative error yielded by NNDSVD-LRC to continue to improve with factorization rank beyond the thresholds at which the other two methods cease to improve, since it uses smaller singular factors and the low rank HALS iterations. The test results are consistent with this expectation. In particular, the initial relative error yielded by NNDSVD-LRC decreases from approximately 0.11 to 0.08 to 0.04 as r changes from 5 to 10 to 15, exhibiting steady decrease with increasing r as expected. NNDSVD yields initial relative errors decreasing from 0.141 to 0.13, and then staying constant at 0.13, as r goes from 5 to 10 to 15, which also makes sense if we assume that the threshold r' is within the range from 10 to 15. The initial relative error generated by SVD-NMF steadily decreases throughout all tests, going from 0.266 to 0.263 to 0.261, suggesting that its threshold r'' is at least 15. Regardless, the fact that in these tests NNDSVD-LRC yields initial relative errors that decrease much faster with increasing factorization rank than both NNDSVD and SVD-NMF points to the success of the low rank correction procedure.

However, even if the low rank correction procedure is the cause of NNDSVD-LRC's rapidly decreasing initial relative error, this does not necessarily make it a more effective initialization procedure than NNDSVD or SVD-NMF as r increases. Recall that to evaluate initializations we must consider the convergence rate and final relative error they cause NMF algorithms to exhibit. The results of the ANLS BPP and PGD tests suggest that the low rank correction method is indeed effective from this broader perspective. All three methods improve a comparable amount when r changes from 5 to 10, but NNDSVD-LRC improves much more than NNDSVD and SVD-NMF when r changes from 10 to 15. However, the results for MU and HALS are not as supportive of the low rank correction procedure. For initializing MU, neither NNDSVD-LRC nor NNDSVD improves significantly in terms of convergence rate and final error yielded relative to the other algorithms with increased factorization rank - both perform poorly for all three tests, likely because they yield sparse initial factors. SVD-NMF performs better than both, but also does not improve relative to the other algorithms as r increases. HALS is the opposite: all three methods improve in performance relative to the other methods as the factorization rank increases, but it is not clear which method improves more than the others. These results are too weak to allow us to definitively say that the low rank correction method causes the effectiveness of NNDSVD-LRC to improve with the rank of the of the factorization more than that of NNDSVD does. At best, we can say that NNDSVD-LRC improves more than NNDSVD for some increments of r and only for initializing some algorithms.

Otherwise, the variation in initialization method performance across all factorization ranks and across the NMF algorithms is very large. Besides the data column subset selection techniques, every initialization technique is among the 2-3 best and/or worst techniques for initializing some algorithm with some factorization rank, making it difficult to describe additional trends in the results.

6.1.4 Binary Data

Since our tests of NMF algorithm performance on binary X and over the condition number of X and the type of heavy-tailed distribution from which the elements of X were selected yielded distinctive results in Chapter 3, we replicate some of those tests using the various initialization techniques in the next three sections. We start with a test on binary data X, in which each element element is 1 with probability 0.5. We plot the error curves for the four NMF algorithms initialized with the various initialization techniques in Figure 6.7.

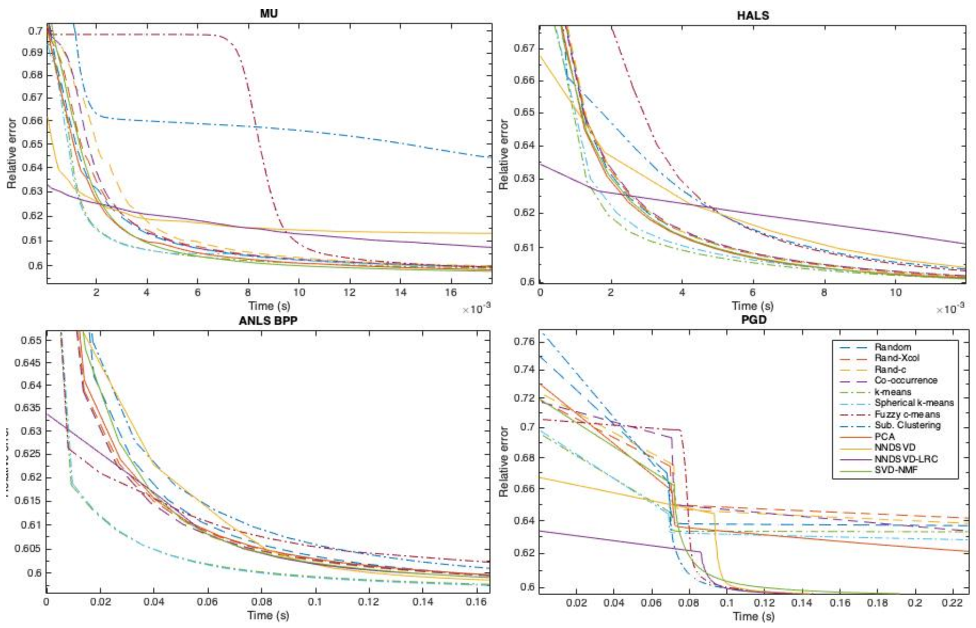


Figure 6.7: NMF relative error vs time for 12 different initialization strategies for (a) MU, (b) HALS, (c) ANLS BPP, and (d) PGD, with X binary with $\mathbf{Pr}[x_{ij} = 1] = 0.5 \forall i, j$.

The results here are similar to those in the previous section. Again, k-means and spherical k-means are the two best or two of the best initialization strategies for initializing MU, HALS, and ANLS BPP. Also, while NNDSVD, NNDSVD-LRC and subtractive clustering are three of the worst algorithms for initializing MU and HALS, they are three of the best for initializing PGD. These results suggest that the binary nature of data does not significantly affect the effectiveness of initialization strategies for NMF algorithms running on it.

6.1.5 Condition Number

We next test initialization performance on datasets with varying condition numbers (in the spectral norm, as usual). To generate a random nonnegative matrix X with a particular condition number, denoted $\kappa(X)$, we follow the same procedure outlined in Section 3.4. The initialization performance results for $\kappa(X) = 20$ and $\kappa(X) = 10,000$ are shown in Figures 6.8 and 6.9.

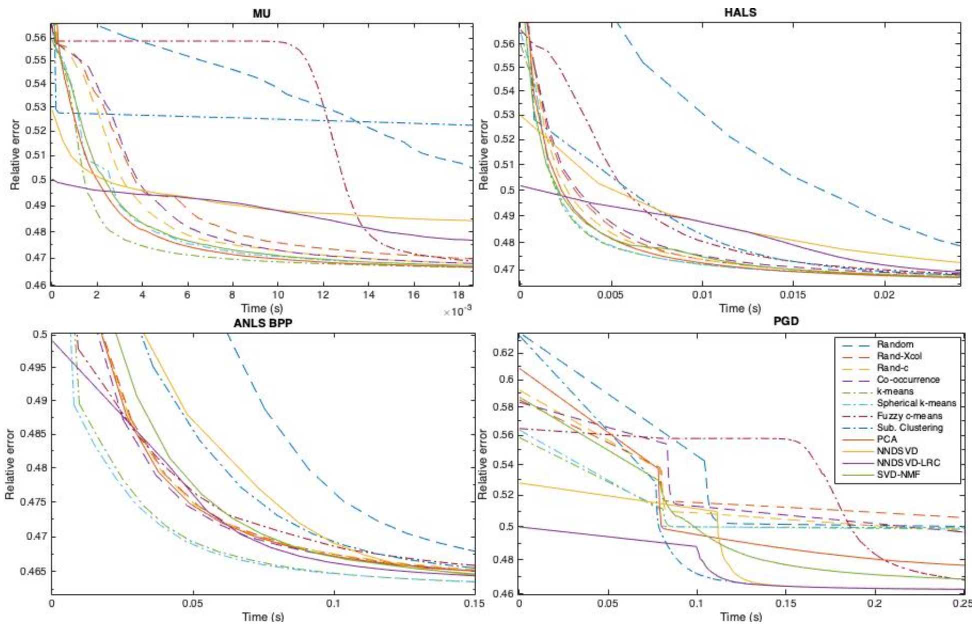


Figure 6.8: NMF relative error vs time for 12 different initialization strategies for (a) MU, (b) HALS, (c) ANLS BPP, and (d) PGD, with $\kappa(X) = 20$.

The results when X has a condition number of 20 are very similar to the results on the vanilla Gaussian data from Section 6.1.1. This makes sense because the data is initially generated in the same way in both cases, and the condition number of the vanilla Gaussian data is 16.5 on average, so the singular values are not scaled by much to generate a matrix with condition number 20. One notable difference is that the random initialization performs significantly worse when the condition number is 20.

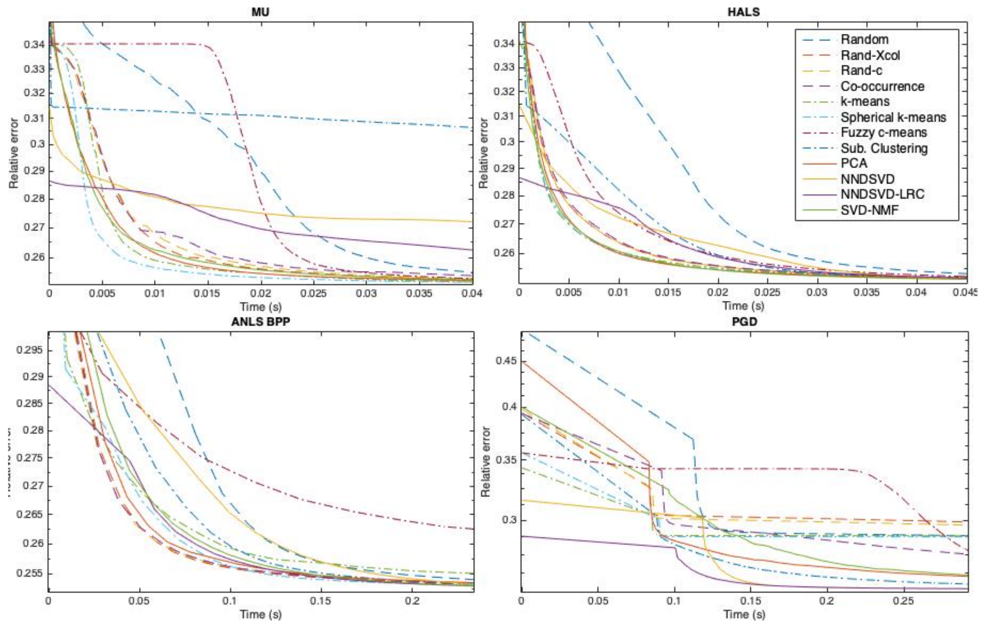


Figure 6.9: NMF relative error vs time for 12 different initialization strategies for (a) MU, (b) HALS, (c) ANLS BPP, and (d) PGD, with $\kappa(X) = 10,000$.

Although the final relative errors for the ill-conditioned matrix test are drastically different (roughly 0.465 for the well-conditioned matrix test and 0.25 for the ill-conditioned matrix test), the results are structurally similar. In both tests, PCA, SVD-NMF, k-means, spherical k-means, and the column subset selection methods perform well for initializing MU, HALS, and ANLS BPP, with the only minor exception being that k-means yields a below average final error when initializing ANLS BPP to factor data with condition number 10,000. Likewise, NNDSVD, NNDSVD-LRC and subtractive clustering perform the best for initializing PGD in both cases. The effectiveness of the low-rank approximation methods on the ill-conditioned data may have to do with the data having fewer non-trivial singular factors, which makes it more conducive to low rank representations. However, the similar effectiveness of these methods on the well-conditioned data belies this conclusion.

6.1.6 Heavy-Tailed Data

In this section we test algorithm performance on data generated according to one of the two heavy-tailed distributions which yielded the most distinctive results from the normal distribution in Section 3.5, the Pareto distribution. As a reminder, the Pareto distribution has density function

$$p(x) = \frac{1}{x^2}, \text{ for } x \geq 1 \tag{6.3}$$

We plot the NMF algorithm error curves for each initialization in Figure 6.10.

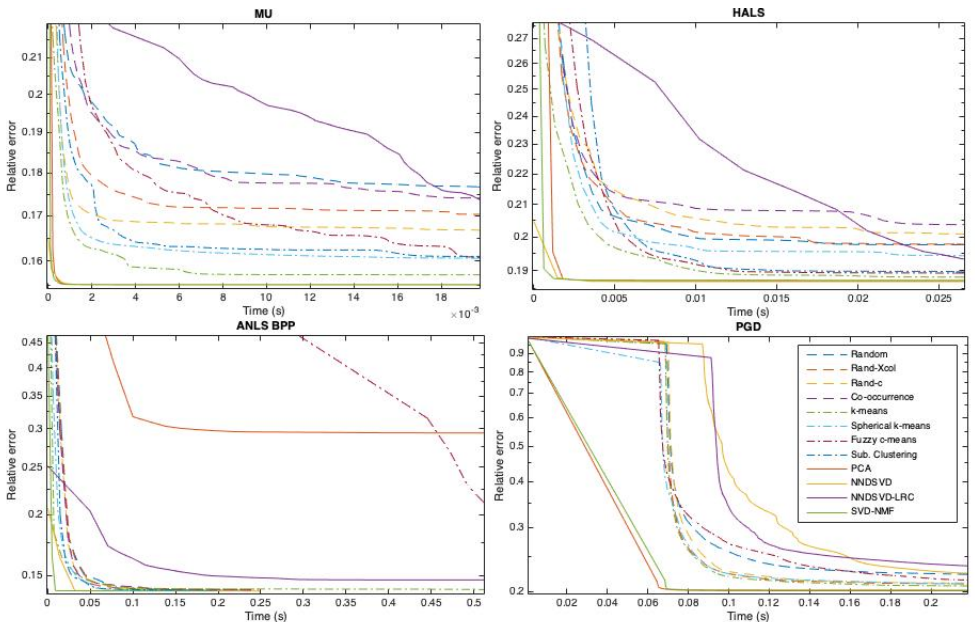


Figure 6.10: X sampled from the Pareto distribution: NMF relative error vs time for 12 different initialization strategies for (a) MU, (b) HALS, (c) ANLS BPP, and (d) PGD.

The exceptional performance of most of the low rank approximation techniques stands out from the results. Specifically, SVD-NMF, NNDSVD and PCA yield error curves that each converge to the lowest relative errors in by far the shortest amount of time for initializing MU and HALS, and SVD-NMF and PCA perform similarly well for initializing ANLS BPP, as do SVD-NMF and NNDSVD for PGD. Our hypothesis for why these methods perform so well is that the r largest singular factors contain proportionally more of the information in this type of heavy-tailed dataset than they do for other datasets, which allows the rank-r approximation methods to represent more of the data than usual, thus allowing the NMF algorithms to reach a more accurate solution more quickly.

Spectral analysis of the heavy-tailed X generated in this section provides the evidence for the fundamental premise of this hypothesis. The largest r singular values of X sum to 72.2% of the sum of the largest 100 singular values of X on average, whereas for the vanilla Gaussian X generated in Section 6.1.1, this measure is only

37.1%. Meanwhile, the sum of the first $\lceil \frac{r}{2} + 1 \rceil$ singular values for the heavy-tailed X is only 51.4% of the sum of the 100 largest singular values of the heavy-tailed X on average, possibly explaining why NNDSVD-LRC does not perform so well in this case. This spectral analysis does not necessarily imply that the rank-r techniques perform exceptionally well here because of the top-heavy spectral profile of X, as there are numerous complications in play, for example the fact that the rank-r approximation techniques do not compute an optimal rank-r approximation to X because of the nonnegativity constraint on the approximation. Nevertheless, the correlation is compelling, and presents an area for future investigation.

6.2 Tests on Real Data

We close this chapter by testing initialization performance on two real-world datasets: the CBCL face dataset from the MIT Center For Biological and Computation Learning[1] and a truncated version of the 1987-2015 NIPS dataset[2] [64]. The CBCL dataset is the dataset used by Lee and Seung in their seminal paper showing that NMF can be used effectively for facial recognition [59], and consists of 2429 front facial images of size 19-by-19 pixels represented as vectors of length 361. Thus, for this dataset the data matrix X has $p = 361$ rows and $n = 2429$ columns. To remain consistent with Lee and Seung's study of this data, we set $r = 49$. Also of note is that this matrix is a dense matrix, as each pixel generally has some nonzero grayscale value. The original 1987-2015 NIPS dataset is a 11462-by-5810 word-by-document matrix of word counts for NIPS conference papers published from 1987 to 2015, but since this matrix is exceptionally large and therefore requires an unreasonably many computational resources, we truncate the matrix to include only the first 1000 words, so it becomes 1000-by-5810. Even with this truncation, the matrix is still much larger than any other matrix we have tested. To factor this matrix we again choose $r = 49$, although this is somewhat arbitrary. Tests on both of these matrices, especially the NIPS dataset, will provide insights as to how the NMF algorithms and initializations perform on large datasets. We display the results of the NMF algorithm executions with the different initializations on the datasets in Figures 6.11 and 6.12.

Before analyzing the results, it is helpful to provide some statistics related to the data. The CBCL dataset is dense, having only 35 of 876869 elements equal to zero. It is also somewhat ill-conditioned with a condition number of approximately 1,568.7.

[1]`http://cbcl.mit.edu/software-datasets/heisele/facerecognition-database.html`
[2]`http://archive.ics.uci.edu/ml/datasets/NIPS+Conference+Papers+1987-2015`

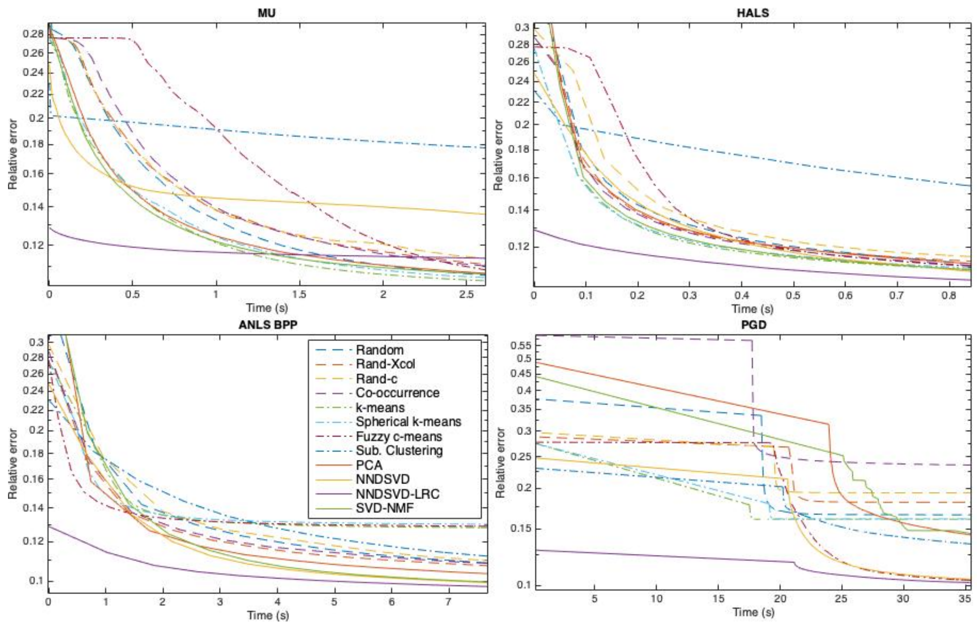


Figure 6.11: X representing the CBCL dataset: NMF relative error vs time for 12 different initialization strategies for (a) MU, (b) HALS, (c) ANLS BPP, and (d) PGD.

The truncated NIPS dataset has the opposite characteristics: it is 93.55% sparse yet relatively well-conditioned with a condition number of 530. Both datasets are full rank.

The CBCL dataset's density yet poor conditioning makes it a somewhat similar dataset to the ill-conditioned matrices that we tested in Section 6.1.5, and the results share numerous similarities. For initializing MU, NNDSVD-LRC starts out as the most accurate method, then over time is surpassed by the hard clustering methods, PCA and SVD-NMF. For initializing HALS, the hard clustering methods are among the best methods in both cases, and as usual, NNDSVD and NNDSVD-LRC perform among the best for initializing PGD. The results do differ in numerous ways as well, most notably in that NNDSVD-LRC clearly is the most effective technique for initializing HALS and ANLS BPP on the CBCL dataset. We can attribute these differences in the results due to the fact that although both datasets are dense and ill-conditioned, the synthetic dataset has a condition number that exceeds the CBCL dataset by a factor of ten. Furthermore, the spectral profile of the CBCL dataset may be especially important.

Indeed, we suspect that NNDSVD-LRC performs so well for the CBCL dataset because its spectral profile is similarly top-heavy as the heavy-tailed dataset we tested in Section 6.1.6. In this case, its $\lceil \frac{r}{2}+1 \rceil$ largest singular values have a proportionally larger weight than other datasets: specifically, the sum of its $\lceil \frac{r}{2}+1 \rceil$ largest singular values is 69.7% of the sum of its 100 largest singular values. In contrast, this measure is only 38.9% for a dataset of the same size whose elements are the absolute values of samples from the standard normal distribution, and only 48.7% for a dataset of the same size with condition number 10,000, generated as in Section 6.1.5. This means that the rank-$\lceil \frac{r}{2} + 1 \rceil$ approximation of the CBCL dataset that NNDSVD-LRC executes captures an abnormally high percentage of the dataset's information, which could be why NNDSVD-LRC is so effective on this dataset. The results in both this section and the previous section suggest the need for further testing of NMF strategies on matrices with various spectral profiles.

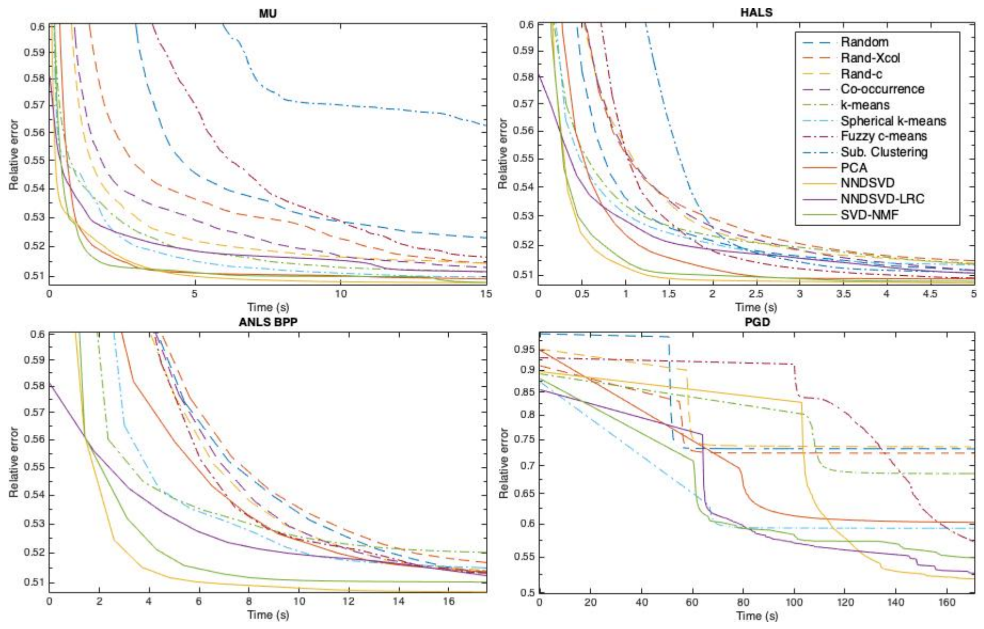


Figure 6.12: X representing the NIPS dataset: NMF relative error vs time for 12 different initialization strategies for (a) MU, (b) HALS, (c) ANLS BPP, and (d) PGD.

We should note that for the large, sparse NIPS dataset, the low rank approximation and clustering initializations each take on the order of seconds to execute, besides subtractive clustering, which takes on the order of hundreds of seconds, likely a sign that it is too inefficient to be used in practice. Meanwhile, the random initialization and the data column subset selection techniques each take on the order of tenths of seconds.

In principle, these results when factoring the NIPS dataset should be similar to the results for the test on data that is 95% sparse that we discussed in Section 6.1.2, since the NIPS dataset is similarly sparse. This principle holds true, as PCA and SVD-NMF perform similarly well for initializing MU and HALS, and SVD-NMF, NNDSVD, and NNDSVD-LRC perform similarly well for initializing ANLS BPP and PGD. One significant difference is that NNDSVD and NNDSVD-LRC perform much better for initializing MU and HALS on the NIPS dataset; perhaps signifying that the sparsity of the initial factors is less detrimental to the performance of MU and HALS when the dataset is large. Also, k-means is substantially more effective for initializing ANLS BPP and PGD for the NIPS dataset than it is for the 95% sparse synthetic data. Also, the column subset selection results have higher variance on the NIPS dataset, which is because we only executed three trials in this case to account for the huge dataset we were dealing with, which is very few if we hoped to mitigate the variance in performance due to the randomness in those techniques. Nevertheless, the superior performance of the low-rank approximation methods is completely deterministic in this setting, and is significantly superior to the data column subset selection techniques, so we do not believe that their superior observed performance is due solely to the small number of trials we executed.

Most importantly, the similarity in the results on the synthetic and real data of similar sparsity is promising. This behavior is intriguing because it suggests that although there is much variation in initialization performance in general - so the technique one chooses to initialize an NMF algorithm with can make a huge difference in algorithm performance - this performance is consistent across similar types of data. Therefore, one can make the critical prediction of how well each initialization will perform on a novel dataset with high accuracy by examining their performances on similar datasets, real or synthetic.

We also observe that the PGD error curves are similar for both the sparse synthetic and real data tests, just as they were for both the ill-conditioned synthetic and real data sets. This behavior is intriguing because it suggests that although there is much variation in initialization performance for PGD - so the technique one chooses to initialize with can make a huge difference in algorithm performance - this performance is consistent across similar types of data. Therefore, one can make the critical prediction of how well each initialization will perform on a novel dataset with high accuracy by examining their performances on similar datasets, real or synthetic.

6.3 Summary of Major Findings

In this section we briefly summarize the major findings from the tests in this chapter.

- The low rank approximation and clustering techniques take an order or two of magnitude longer to execute than the random initialization and the data column subset selection methods, although they generally yield more effective initializations.

- The performance of PGD varies the most across the choice of initialization method, while the performance of ANLS BPP varies the least.
- The more sparse the initial factors, the less favorable error curves they yield for MU and HALS. This means that NNDSVD and NNDSVD-LRC typically perform poorly for initializing these algorithms.
- k-means and spherical k-means generally perform very well for initializing MU, HALS, and ANLS BPP when the data is Gaussian, moderately sparse, well-conditioned, and binary. Conversely, for very sparse, ill-conditioned and heavy-tailed datasets, especially those with 'top-heavy' spectral profiles, at least some of the low rank techniques perform the best for initializing MU, HALS, and ANLS BPP on these datasets. Meanwhile NNDSVD, NNDSVD-LRC and at least one of the soft clustering techniques typically perform very well for initializing PGD, regardless of the type of data we tested.
- Nevertheless, relative initialization performance varies significantly across the different qualities of data we tested. We refer the reader to our test results for more precise conclusions.
- Despite the variation in initialization performance across different qualities of data, performance is consistent across the synthetic 95% sparse dataset and the NIPS dataset, which is similarly sparse, a promising sign for the usefulness of testing NMF strategies on synthetic datasets in order to predict their effectiveness on real datasets of similar characteristics.

Chapter 7

Recent Algorithms

All of the the NMF algorithms we have discussed so far have been around for quite some time: the most recent ones, the active set [52] and block principal pivoting [53] techniques for alternating nonnegative least squares, were first introduced by Kim and Park in 2008. While these algorithms remain popular in practice, in the past 8 years many algorithms have been developed that strive to improve on the fundamental NMF algorithms we have already discussed. Some are iterative heuristics with no provable performance guarantees, while others come with novel guarantees on the accuracy of the factors they produce under certain assumptions. These algorithms require a thorough, comparative empirical analysis to determine the extents to which they should be used in practice. Many of them have yet to be thoroughly evaluated against each other experimentally, not to mention against the fundamental algorithms we discussed earlier, so this work aims to help fill this gap in the literature. We start by introducing each of the recent algorithms in this chapter, then test them on both synthetic and real data in the subsequent chapter.

7.1 Iterative Heuristics

We first discuss two recent iterative algorithms which, like the seven algorithms we discussed in Chapter 2, are not provably accurate or efficient, but may work well in practice.

7.1.1 Alternating Direction Method of Multipliers

The alternating direction method of multipliers (ADMM) is a fundamental algorithm used to tackle many optimization problems. To the best of our knowledge, Xu et.

al. were the first to apply ADMM to NMF [79]. They did so in 2012, then Sun and Févotte later extended their work to apply to all NMF objective functions in the beta-divergence class [68].

ADMM attempts to solve an optimization problem by separating it into smaller optimization problems, each of which may be easier to solve [17]. The problem it tries to solve is of the form:

$$\min_{x \in \mathcal{X}_i \forall i=1,2,\ldots,q} \sum_{i=1}^{q} g(x_i); \quad \text{s.t.} \; \sum_{i=1}^{q} \mathcal{A}_i(x_i) = c \tag{7.1}$$

where each of the g_i's are functions defined on the closed subsets $\mathcal{X}_i$ of a finite-dimensional space, and the $\mathcal{A}_i$'s are linear operators and c is an appropriately sized matrix or vector. The augmented Lagrangian of 7.3 is

$$\mathcal{L}((x_1, ..., x_q), \lambda) = \sum_{i=1}^{q} g_i(x_i) + \lambda^T \Big(\sum_{i=1}^{q} A_i x_i - b \Big) + \frac{\rho}{2} \|A_i x_i - b\|_2^2 \tag{7.2}$$

where λ is the Lagrangian multiplier vector and ρ is a penalty parameter. The ADMM method is an iterative procedure that separately updates each x_i by choosing them to minimize $\mathcal{L}$, then updates λ according a gradient ascent on every iteration.

The nonnegativity constraints on W and H pose issues when applying this framework to NMF, since there are no inequality constraints in the problem formulation associated with ADMM. To overcome this, the NMF optimization problem in the Euclidean norm is reformulated as follows [68]:

$$\begin{aligned} \min_{W \in \mathbb{R}^{p \times r}, H \in \mathbb{R}^{r \times n}} \quad & \textit{f(W,H)} \\ \text{s.t.} \quad & W = W_+, H = H_+ \\ & W_+ \geq 0, H_+ \geq 0 \end{aligned} \tag{7.3}$$

where the objective function f remains the same:

$$f(W, H) = \frac{1}{2} \|X - WH\|_F^2 \tag{7.4}$$

In the ADMM algorithm, we can force W_+ and H_+ to be nonnegative, then implement the nonnegativity constraints on W and H by penalizing their distance from W_+, H_+ within the update procedure. Thus, there are four primal and two dual variables to optimize. Yet from the ADMM perspective, which splits larger optimization problems into smaller ones, this optimization problem is a three-block optimization problem, with the blocks optimizing over W, H, and (W_+, H_+). (W_+, H_+) is one optimization block because the optimization problem never puts both W_+ and H_+ in the same constraint or same objective function, so optimizing both separately is equivalent to optimizing both jointly [68]. The augmented Lagrangian associated with the optimization problem is:

$$\begin{aligned}\mathcal{L}(W, H, W_+, H_+, \lambda_W, \lambda_H) =& \frac{\rho}{2}\|X - WH\|_F^2 + \langle \lambda_W, W - W_+ \rangle + \frac{\rho}{2}\|W - W_+\|_F^2 \\ &+ \langle \lambda_H, H - H_+ \rangle + \frac{\rho}{2}\|H - H_+\|_F^2 \end{aligned} \tag{7.5}$$

ADMM updates the primal variables to minimize the Lagrangian on the t^{th} iteration as follows:

$$W^{(t)} = (XHW^{(t-1)^T} + W_+W^{(t-1)} - \frac{1}{\rho}\lambda_W^{(t-1)})(H^{(t-1)}H^{(t-1)^T} + I)^{-1} \tag{7.6}$$

$$H^{(t)} = (W^{(t)^T}W^{(t)} + I)^{-1}(W^{(t)^T}X + H_+^{(t-1)} - \frac{1}{\rho}\lambda_H^{(t-1)}) \tag{7.7}$$

$$W_+^{(t)} = \mathcal{P}_{NN}(W^{(t)} + \frac{1}{\rho}\lambda_W^{(t-1)}) \tag{7.8}$$

$$H_+^{(t)} = \mathcal{P}_{NN}(H^{(t)} + \frac{1}{\rho}\lambda_H^{(t-1)}) \tag{7.9}$$

where as previously, $\mathcal{P}_{NN}(A)_{ij} = \max(a_{ij}, 0)$ [68]. Because of the addition of the identity matrix, the matrices to be inverted when updating W and H are invertible. ADMM also updates the dual variables on the same iteration [68]:

$$\lambda_W^{(t)} = \lambda_W^{(t-1)} + \rho(W^{(t)} - W_+^{(t)}) \tag{7.10}$$

$$\lambda_H^{(t)} = \lambda_H^{(t-1)} + \rho(H^{(t)} - H_+^{(t)}) \tag{7.11}$$

and repeats this procedure on every iteration until some stopping condition is satisfied [68]. Since we care more about observing the evolution of the algorithm's error curve over time than we do about executing the minimum sufficient number of iterations, in our implementation we use a maximum on the total number of iterations allowed instead of a more efficient stopping condition.

Each of the primal and dual variables needs to be initialized. In accordance with [79], we initialize the dual variables and $W_+^{(0)}$ and $H_+^{(0)}$ as matrices of zeros and $W^{(0)}$ and $H^{(0)}$ using the random initialization described in Chapter 5. A variety of values for ρ may be used; after testing a variety of values we find that $\rho = 1$ is generally the most effective. A summary of the algorithm is given in Algorithm 11.

Algorithm 11: ADMM [68]

```
**Input:** Nonnegative matrix $X \in \mathbb{R}^{p \times n}$ and factorization rank $r$
**Output:** Nonnegative matrices $W_+ \in \mathbb{R}^{p \times r}$ and $H_+ \in \mathbb{R}^{r \times n}$ such that $X \approx WH$
  Initialize $(W^{(0)}, H^{(0)}, W_+^{(0)}, H_+^{(0)}, \lambda_W^{(0)}, \lambda_H^{(0)})$
  Set $\rho > 0$, $t \leftarrow 1$
  while Stopping criteria not satisfied do
      Update $(W^{(t)}, H^{(t)}, W_+^{(t)}, H_+^{(t)}, \lambda_W^{(t)}, \lambda_H^{(t)})$ according to Equations 7.6 and 7.10
      $t \leftarrow t + 1$
  end while
  return $W_+^{(t)}, H_+^{(t)}$
```

ADMM is known to converge when it is applied to two-block convex programs, corresponding to the scenario when $q = 2$ and g_1 and g_2 are convex in 7.3 [68]. Otherwise, no convergence guarantees are known [68]. Since in the NMF setting we are dealing with a three-block optimization problem with a nonconvex objective function, the convergence guarantees of ADMM do not apply. Nevertheless, Xu et. al. provide qualified performance guarantees for ADMM. Specifically, they show that if ADMM for NMF converges, then it converges to a point which satisfies the first-order necessary conditions for optimality (the KKT conditions) [79]. However, results have not been attained that shows that ADMM for NMF converges to a globally optimal point, the rate at which it converges, or that it even converges at all.

The runtime of this algorithm depends on the cost of the matrix inversions and multiplications required to update the variables during each iteration. The two matrix inversions are both of r-by-r matrices, so they can be done in $O(r^3)$ iterations or better. Each matrix multiplication requires no more than $O(prn)$ multiplications, which is larger than $O(r^3)$ under our assumption that $r \ll p, n$. So the total cost per iteration is $O(prn)$, equivalent to the NMF algorithms we have studied previously.

7.1.2 Low Rank Approximation-Based NMF (lraNMF)

The next NMF algorithm we discuss is motivated by the same principle that inspired the low rank correction procedure within Syed and Gillis' NNDSVD-LRC initialization method. That is to reduce the computational cost of NMF algorithm iterations while maintaining the majority of their effectiveness by running them with a low rank approximation of X as the target matrix. This NMF with low rank approximation technique was formally proposed by Zhou and Cichocki in 2012, who called their method low rank approximation-based NMF (lraNMF) [84].

To provide more formal intuition for their technique, they reframe the NMF objective function as one with two Frobenius norms:

$$\begin{aligned} \min \quad & F(\tilde{W}, \tilde{H}, W, H) = \|X - \tilde{W}\tilde{H}\|_F^2 - \|\tilde{W}\tilde{H} - WH\|_F^2 \\ \text{s.t.} \quad & \tilde{W} \in \mathbb{R}^{p\times l}, \tilde{H} \in \mathbb{R}^{l\times n},\ W \in \mathbb{R}_{\geq 0}^{p\times r}, H \in \mathbb{R}_{\geq 0}^{r\times n} \end{aligned} \tag{7.12}$$

where $l = kr \ll \min(p, n)$, and k is a small positive constant [84]. This objective function implies a two-step solution [84]:

1. Unconstrained rank-l approximation of X to minimize $\|X - \tilde{W}\tilde{H}\|_F^2$.
2. Nonnegative matrix factorization of the fixed matrix $\tilde{W}\tilde{H}$ to minimize $\|\tilde{W}\tilde{H} - WH\|_F^2$ subject to the constraints that W and H are nonnegative.

The first step can be solved by any low rank approximation technique. We use the truncated SVD because it provably minimizes the objective $\|X - \tilde{W}\tilde{H}\|_F^2$ with $\tilde{W}\tilde{H}$ having inner dimension l. Specifically, if $[U, \Sigma, V]$ is the rank-l truncated SVD of X, we set $\tilde{W} = U\sqrt{\Sigma}$ and $\tilde{H} = \sqrt{\Sigma}V^T$. Note that $\tilde{W}$ and $\tilde{H}$ have roughly 50% negative entries under this construction.

The second step may be tackled via any of the NMF algorithms we have discussed which can handle the target matrix not being nonnegative. Zhou and Cichocki suggest and implement MU and HALS for this step in separate algorithms because of their popularity [84]. For the sake of space, we only implement one of these sub-algorithms, and choose to implement the HALS version (lraNMF-HALS) because of the generally superior performance of HALS over MU that we have observed in Chapter 2, and because MU already has smaller runtime per iteration, meaning that HALS has more room to grow in this area. Moreover, MU requires a modification to ensure that its estimates of W and H converge to a stationary point when X has negative elements, whereas the convergence of the HALS iterations to a stationary point is guaranteed regardless of whether X is nonnegative or not [84].

Now having $\tilde{W}\tilde{H}$ as the target matrix, the update rules for HALS become:

$$W(:, j) \leftarrow \max\left(0, \frac{\tilde{W}\tilde{H}H(j,:)^T - \sum_{k\neq j} W(:,k)(H(k,:)H(j,:)^T)}{\|H(j,:)\|_2^2}\right) \tag{7.13}$$

$$H(j, :) \leftarrow \max\left(0, \frac{\tilde{H}^T\tilde{W}^TW(:,j) - \sum_{k\neq j} H(k,:)(W(:,k)^TW(:,j))}{\|W(:,j)\|_2^2}\right) \tag{7.14}$$

The previous computational bottleneck for each column and row update was multiplying $XH(j,:)^T$ and $X^TW(:,j)$, respectively, which required $O(pn)$ operations. Now, those computations have been replaced by $\tilde{W}\tilde{H}H(j,:)^T$ and $\tilde{H}^T\tilde{W}^TW(:,j)$, which only require $O((p+n)l) = O((p+n)kr)$ operations. Thus, the total cost per iteration reduces from $O(prn)$ to $O((p+n)kr^2)$ operations, which simplifies to $O((p+n)r^2)$ operations when $k=1$, which is a typical choice and the choice we use in our implementation. Note that computing the truncated rank-l SVD of X still requires $O(prn)$ iterations, but since this is a one-time cost, it is not as important as the cost per iteration.

Naturally we may wonder what the cost of this speedup is in terms of accuracy. Zhou and Cichocki provide error bounds which quantify this cost, which we summarize in Proposition 2.

Proposition 2. *[84] Given the data matrix $X \in \mathbb{R}^{p\times n}$, suppose there exist nonnegative matrices $W^\natural$ and $H^\natural$ such that $(W^\natural, H^\natural) = \arg\min_{W\in\mathbb{R}_+^{p\times r}, H\in\mathbb{R}_+^{r\times n}} \|X - WH\|_F^2$, and let $e^\natural = \|X - W^\natural H^\natural\|_F$. Furthermore, let $\tilde{X} = \tilde{W}\tilde{H}$ be some low rank approximation to X. Then:*

1. $\min_{W\geq 0, H\geq 0, \tilde{X}\mathbb{R}^{p\times n}} \|X - \tilde{X}\|_F + \|\tilde{X} - WH\|_F = e^\natural$
2. *If $\|X - \tilde{X}\|_F = \sigma$, and there exist matrices W^* and H^* such that $(W^*, H^*) = \arg\min_{W\geq 0, H\geq 0} \|\tilde{X} - WH\|_F$, then $e^\natural \leq \|X - W^*H^*\|_F \leq 2\sigma + e^\natural$.*

To prove (1), one can use the Triangle Inequality to show that the objective function is at least $\|X - W^\natural H^\natural\|_F$, then set $W = W^\natural, H = H^\natural$, and $\tilde{X} = W^\natural H^\natural$ and observe that this assignment of variables causes the objective function to achieve its lower bound [84]. To prove (2), the lower bound on $\|X - W^*H^*\|_F$ follows straightforwardly from the definition of $e^\natural$, and the upper bound follows from two applications of the Triangle Inequality [84]:

$$\begin{aligned}
\|X - W^*H^*\|_F &\leq \|X - \tilde{X}\|_F + \|\tilde{X} - W^*H^*\|_F && (7.15)\\
&= \sigma + \|\tilde{X} - W^*H^*\|_F && (7.16)\\
&\leq \sigma + \|\tilde{X} - W^\natural H^\natural\|_F && (7.17)\\
&\leq \sigma + \|\tilde{X} - X\|_F + \|X - W^\natural H^\natural\|_F && (7.18)\\
&= 2\sigma + e^\natural && (7.19)
\end{aligned}$$

These results are important because they show that the optimal solutions to the original NMF problem and to the lraNMF problem have the same error in terms of their own respective objective functions (1), and that the optimal low rank NMF has error that is within a scalar multiple of the low rank approximation error from the optimal NMF of X (2). The latter point tells us that the smaller we make the low rank approximation error, the closer we know that an lraNMF solution exists to the optimal NMF solution. Yet this guarantee comes with the higher computational cost per iteration associated with a larger l, since to obtain a more accurate low rank approximation of X, we have to increase l. For simplicity, we hold l equal to r in our implementation and experiments.

7.2 Near-Separable NMF

We now move on to algorithms which have performance guarantees under particular assumptions on the data. Arora et al. sparked a research shift from practical heuristics towards provably efficient and accurate algorithms with their 2012 paper showing that a particular type of NMF problem can be solved in polynomial time [4]. Their work, like many of its successors and all of the algorithms we discuss in this section, relies on the assumption that X is *separable*[1]:

Definition 3. *[37] A matrix X is r-**separable** if and only if there exists a subset of r of the columns of X which each of the other columns of X is a positive linear combination of, i.e. they are contained within the convex cone formed by the vectors in the subset. In other words, X is r-separable if and only if there exists an index set $\mathcal{K}$ of cardinality r and a nonnegative matrix H such that*

$$X = X(:, \mathcal{K})H \tag{7.20}$$

where we use MATLAB notation to specify the submatrix of X indexed by $\mathcal{K}$.

The problem of computing the index set $\mathcal{K}$ is known as separable NMF [37]. Finding $\mathcal{K}$ is equivalent to finding the basis matrix W, and once this has been found, H can be computed by solving the nonnegative least-squares problem.

If each column of X is normalized such that its elements sum to one, namely $X(:, j) \leftarrow \|X(:, j)\|_1 X(:, j)$ for all $j = 1, 2, ..., n$, then each of the columns of H must also sum to 1 [37]. To see this, note that for all j:

[1]Some make the more general assumption of *near-separability*, we will discuss this later.

$$\begin{aligned}
1 &= \|X(:,j)\|_1 && (7.21)\\
&= \|X(:,\mathcal{K})H(:,j)\|_1 && (7.22)\\
&= \sum_{i=1}^{r} \|X(:,\mathcal{K}(i))H(i,j)\|_1 && (7.23)\\
&= \sum_{i=1}^{r} \|X(:,\mathcal{K}(i))\|_1 H(i,j) && (7.24)\\
&= \sum_{i=1}^{r} H(i,j) && (7.25)\\
&= \|H(:,j)\|_1 && (7.26)\\
& && (7.27)
\end{aligned}$$

where the third, fourth, and last inequalities follow by the nonnegativity of the elements [37]. Thus, each normalized column of X is a convex combination of the columns of $X(:,\mathcal{K})$, so computing $\mathcal{K}$ amounts to finding the convex hull of the columns of X. We will discuss algorithms that will exploit this fact later.

Separable NMF has been studied for quite some time and in many contexts, including:

- In text mining, or topic modelling from word-by-document matrices, the separability assumption entails that for each topic, there is some document which contains only that topic. However, a more realistic assumption is that for each topic, there exists a word which is used in only that topic. This means that the transposed word-by-document matrix (the document-by-word matrix) would be separable. Such words are known as "anchor" words [4].

- In hyperspectral unmixing, the wavelength-by-pixel matrix is separable if and only if for each endmember there is a pixel containing only that endmember, which is an assumption used since the 1990s [45].

- In numerical linear algebra, separable NMF is closely related to the well-known problem of column subset selection [37].

These examples suggest that separability is a realistic assumption to make. Still, in some contexts the presence of noise means that separability does not apply; in these settings, the *near-separable NMF* problem is formulated as:

Definition 4. *A matrix $\tilde{X}$ is known as **near-separable** if it is the sum of a separable matrix and noise. Namely, $\tilde{X} = X + N$ with $X = W^{\natural}[I_r, H^{\natural'}]\Pi$, where $W^{\natural}$ and $H^{\natural'}$ are nonnegative matrices, I_r is an r-by-r identity matrix, Π is a permutation matrix and N is the noise. The near-separable NMF problem is to find a set $\mathcal{K}$ of r indices such that $\tilde{X}(:, \mathcal{K}) \approx W$.*

Note that the permutation matrix is present to allow the identity matrix I_r to be expressed, showing that X is r-separable. Here, $H = [I_r, H^{\natural'}]\Pi$. Also note that the near-separable NMF problem is a generalization of the separable NMF problem.

A variety of algorithms have been presented to solve near-separable NMF, and they generally fall into one of two categories: those that use geometric intuition and the successive projection algorithm framework (such as [2, 3]), and those that use linear programming to solve a self dictionary and sparse regression problem (such as [40, 12]). We will start by discussing the former category of algorithms, then move on the the latter. Keep in mind that we are introducing these algorithms with the larger goal of testing their performance, which we will do in the next chapter on both near-separable and non-near-separable datasets.

7.2.1 Successive Projection Algorithm

The successive projection algorithm (SPA) forms the foundation for most popular geometric algorithms to provably solve near-separable NMF [37]. Here we will discuss the algorithm, and in the next section we will examine one of its close relatives, the FastAnchorWords algorithm [3].

SPA aims to solve the near-separable NMF problem by finding the vertices of the convex hull of the columns of $\tilde{X}$, and its means of doing so is simple yet effective. On each of r iterations, it projects each of the columns of $\tilde{X}$ onto the orthogonal complement of the current longest column, and greedily adds the index of the longest column to $\mathcal{K}$. In addition to the near-separability assumptions, it assumes that $W^{\natural}$ is full rank and the elements of the columns of $H^{\natural'}$ sum to at most one (otherwise, they can be normalized to do so without loss of generality). Note that the full rank assumption is necessary or else the algorithm would project some unidentified column(s) of W to zero, preventing them from later being identified. The algorithm is outlined in Algorithm 12.

Algorithm 12: Successive Projection Algorithm [2]

```
Input: Near-separable matrix $\tilde{X} = W^{\natural}[I_r, H^{\natural\prime}]\Pi + N$, where $W^{\natural}$ is full-rank,
  $(H^{\natural})' \geq 0$, the entries of each column of $H^{\natural\prime}$ sum to at most 1, $\Pi$ is a permutation
  matrix, and $N$ is the noise; factorization rank $r$.
Output: Set $\mathcal{K}$ of $r$ indices such that $\tilde{X}(:, \mathcal{K}) \approx W^{\natural}\Pi'$ for some permutation $\Pi'$.
  Let $R = \tilde{X}$, $\mathcal{K} = \{\}$
  for $k = 1$ to $r$ do
      $q = \arg\max_j \|R_{:,j}\|_2$
      $R = \big(I - \frac{R_{:,q}R_{:,q}^T}{\|R_{:,q}\|_2^2}\big)R$
      $\mathcal{K} = \mathcal{K} \cup \{q\}$
  end for
  return $\mathcal{K}$
```

On each iteration, this algorithm requires $O(pn)$ operations to update R, noting that each of the n columns of R can be updated using the formula:

$$R_{:,j} \leftarrow \Big(I - \frac{R_{:,q}R_{:,q}^T}{\|R_{:,q}\|_2^2}\Big)R_{:,j} = R_{:,j} - \frac{R_{:,q}}{\|R_{:,q}\|_2^2}\Big(R_{:,q}^T R_{:,j}\Big) \tag{7.28}$$

Furthermore, $\arg\max_j \|R_{:,j}\|_2$ can be computed in only n operations using the formula [37]:

$$\Big\|\Big(I - \frac{R_{:,q}R_{:,q}^T}{\|R_{:,q}\|_2^2}\Big)R_{:,j}\Big\|_2^2 = \|R_{:,j}\|_2^2 - \frac{1}{\|R_{:,q}\|_2}\Big(R_{:,q}^T R_{:,j}\Big)^2 \tag{7.29}$$

and the fact that $\frac{1}{\|R_{:,q}\|_2}\big(R_{:,q}^T R_{:,j}\big)$ was computed on the previous iteration. Thus the algorithm runs in only $O(prn)$ operations with a small constant, making it the same complexity as the NMF heuristics but likely faster in practice.

SPA has been shown to correctly solve separable NMF. Here we explain the proof given by Gillis in [37].

Proposition 5. *[37] SPA exactly identifies the vertices of the convex hull of X when no noise is present and $W^{\natural}$ is full rank.*

Proof. The proposition can be shown using induction on the iteration number k. The base case is satisfied because the point with maximum ℓ_2 norm among a set of points must be a vertex of that set's convex hull, because any convex combination of the rest of the points will yield a point with strictly smaller ℓ_2 norm.

For the induction step, assume that SPA has found the first k columns of W (all of which are vertices of the convex hull of the columns of X) and let $W_k = W(:, 1 : k)$ and $P^{\perp}_{W_l}$ be the projection onto the orthogonal complement of the columns of W_k. Note that after k iterations of SPA, $R = P^{\perp}_{W_l} X$. We have that for all $j = 1, 2, ..., n$,

$$\begin{aligned} \|R(:, j)\|_2 = \|P^{\perp}_{W_k} X(:, j)\|_2 = \|P^{\perp}_{W_k} WH(:, j)\|_2 &\leq \sum_{l=1}^{r} H(l, j)\|P^{\perp}_{W_k} W(:, j)\|_2 & (7.30) \\ &= \sum_{l=k+1}^{r} H(l, j)\|P^{\perp}_{W_k} W(:, j)\|_2 & (7.31) \\ &\leq \max_{k+1 \leq j \leq r} \|P^{\perp}_{W_k} W(:, j)\|_2 & (7.32) \end{aligned}$$

where the first inequality follows from the Triangle Inequality, and the second because each element of H is greater than 0 and $\sum_{l=k+1}^{r} H(l, j) \leq 1$ for all j. Since the columns of W are linearly independent and the ℓ_2 norm is strongly convex, the first inequality is strict unless $H(l, j) = 1$ for some l. This means that $X(:, j) = W(:, l)$. If this l is less than or equal to k, then j has already been selected as an index of a convex hull vertex vector and $\|R(:, j)\|_2 = 0$. Otherwise, $\|R(:, j)\|_2 = \|P^{\perp}_{W_k} W(:, j)\|_2 > 0$. By our assumptions, we know that there exists an $l \geq k + 1$ and a j such that $H(l, j) = 1$, so an each iteration k, the largest $\|R(:, j)\|_2$ achieves $\max_{k+1 \leq l \leq r} \|P^{\perp}_{W_k} W(:, l)\|_2$, so for the j for which $\|R(:, j)\|_2$ is maximum, $X(:, j) = W(:, l)$. By definition of W the index j added to the index set $\mathcal{K}$ is the index of a vertex of the convex hull of the columns of X [37]. □

In the presence of noise, if $\epsilon = \max_j \|N(:, j)\|_2 \leq O\big(\frac{\sigma_{\min}(W)}{\sqrt{r}\kappa^2(W)}\big)$, it can be shown that SPA identifies the columns of W from $\tilde{X}$ with ℓ_2 error $O(\epsilon\kappa^2(W))$, making it robust to noise [43]. Nevertheless, numerous variations of SPA have been proposed which lessen the error due to noise. These include:

- filtering out the noise by computing a low rank approximation of $\tilde{X}$ using techniques such as PCA [63];
- preconditioning $\tilde{X}$ to reduce its condition number by projecting it onto the minimum volume ellipsoid containing all of its columns [44, 61];
- executing a more efficient projection by taking advantage of the nonnegativity constraints [36];
- solving the NNLS problem for H on every iteration, then updating R as the residual matrix of $\tilde{X}$ given the current factorization [56];

- refining the current index set on every iteration by determining whether each computed vertex is still satisfactory, and replacing it if not [3].

We will focus on the last variation, the FastAnchorWords algorithm, in the next section.

7.2.2 FastAnchorWords

The FastAnchorWords algorithm was introduced by Arora et al. in 2013 [3] as an improvement to the AnchorWords algorithm introduced by Arora et al. in 2012 [4]. Both algorithms were introduced in the specific context of topic modelling from document-by-word matrices, hence their names, and indirectly solve the near-separable NMF problem under the influence of noise. FastAnchorWords sacrifices some of the speed of SPA for greater robustness to noise, and is now considered one of the state-of-the-art algorithms for near-separable NMF, especially in terms of theoretical guarantees [45].

Before explaining the FastAnchorWords algorithm, we put it and the AnchorWords algorithm in their algorithmic contexts. Both algorithms assume that the data is generated from a probabilistic topic model, similar to the Gaussian model Desmarais used to model the question-by-student data [25], as we analyzed in Chapter 4. Their probabilistic model and problem formulation does not exactly match the near-separable NMF framework, but we will show how they can be reinterpreted to be consistent with the framework.

The authors assume that the data they observe is a corpus of documents with words in them, and their goal is to recover the underlying distribution of words in each topic in the documents [3]. Making use of prior notation, we note that in the AnchorWords and FastAnchorWords setup there are p documents, r topics, and n words. $U \in \mathbb{R}^{p \times r}$ is the (randomly generated) document-by-topic matrix, and $V \in \mathbb{R}^{r \times n}$ is the (fixed) topic-by-word matrix. The authors do not mention a document-by-word matrix, but by observing the document corpus, they observe all the information that would be contained in the document-by-word matrix. They assume the data is generated in the following manner: each document j is randomly assigned a topic distribution $u_{:,j} \in \mathbb{R}^r$ (such as from a Dirichlet or log-normal distribution), then each position (where a word would be) in the document is assigned a topic z from the distribution $u_{:,j}$, and the particular word is chosen from $v_{z,:} \in \mathbb{R}^n$, the distribution over words for the topic z. They also assume separability: for each topic, there is one word that only appears in that topic, i.e. one column of V which has only one nonzero element [3].

In [4] and [3], the authors emphasize that since U is generated stochastically, it cannot be perfectly recovered. After using AnchorWords and FastAnchorWords, respectively, to recover the anchor words (the words used only by one topic), they present separate procedures to recover the fixed topic-by-word matrix V and the topic-by-topic covariance matrix $U^T U$ without ever explicitly recovering U [3].

For our implementation, we reformulate this setup in the near-separable NMF framework. We assume the noisy document-by-word matrix $\tilde{X} = W^\natural[I_r, H^{\natural'}]\Pi + N$ is observed, where $W^\natural$ represents U, $[I_r, H^{\natural'}]\Pi$ represents V, and the noise N represents the distance between the random samples that compose the data and their expectations. Clearly, the columns of the identity matrix I_r correspond to the anchor words. This formulation entails that $W^\natural$ has columns equal to the document profiles of the anchor words, which happen to be specified by the outputs of FastAnchorWords (and AnchorWords). To recover $H \coloneqq [I_r, H^{\natural'}]\Pi$, we use the same procedure we use in our implementation of the SPA algorithm - solving the NNLS subproblem in H with block principal pivoting (BPP).

Like SPA, FastAnchorWords and AnchorWords identify anchor words by finding the vertices of the convex hull of the rows or columns of some matrix, but here that matrix is the empirical word co-occurrence matrix $\tilde{Q}$, not the document-by-word matrix $\tilde{X}$. In the case of infinite data, or in our model no noise, $Q = V^T\mathbb{E}[U^T U]V \equiv H^{\natural^T} W^{\natural^T} W^\natural H^\natural$, but for real settings the empirical (i.e. noisy) word co-occurrence matrix is computed as $\tilde{Q} = \tilde{X}^T\tilde{X}$. Using simple algebra and the definition of the elements of Q within the probabilistic topic modelling framework, it is straightforward to show that the the rows of Q corresponding to anchor words form a convex hull which all the other rows of Q belong to [3]. Thus, finding the row indices of the vertices of the convex hull of the rows of Q is equivalent to finding the column indices of X which are columns in W. Furthermore, it is reasonable to expect that the vertices of the convex hull of the rows of the noisy $\tilde{Q}$ will be close to the vertices of the convex hull of the rows of Q, so identifying those vertices should be sufficient [3].

To the best of our knowledge, AnchorWords was the first algorithm to provably solve near-separable NMF (via the probabilistic topic modelling framework) in polynomial time [37]. However, the algorithm is impractical, as it requires solving a linear program for each row of $\tilde{Q}$ to determine whether it is a vertex of the convex hull [4]. FastAnchorWords is a significantly faster algorithm in practice, yet retains the performance guarantees of AnchorWords. To do so it follows a similar format as SPA.

Yet the differences with SPA, besides that it computes the convex hull of a different set of vectors, are twofold [37, 3]:

- FastAnchorWords first projects all the vectors onto a randomly chosen linear subspace of $\mathbb{R}^{4\log(n)/\epsilon^2}$, so the later projections are onto the affine hull of the vectors extracted so far, not the linear span;

- after extracting all the estimated vertices of the convex hull, FastAnchorWords refines the set of estimates using the following post-processing step:

  – Let $\mathcal{K}$ be the set of indices of the extracted vectors. For each $k \in \mathcal{K}$:

    1. Compute the projection of R onto the orthogonal complement of $R(\mathcal{K} \setminus \{k\}, :)$
    2. Replace k with the index of the column of R with maximum ℓ_2 norm

An overview of this algorithm is given in Algorithm 13. Note that the vectors exist in $\mathbb{R}^{4\log(n)/\epsilon^2}$ after the initial projection, where the authors recommend choosing $1/\epsilon = 1000$. The post-processing step of this algorithm can be implemented in $O(nr\log(n)/\epsilon^2)$ steps, equal to the complexity of the first round of projections. The initial projection takes $O(n^2 + n\log(n)/\epsilon^2)$ time, so the total runtime is

$O(n^2 + nr\log(n)/\epsilon^2)$, significantly larger than SPA and the NMF heuristics because ϵ is assumed to be small [3].

Algorithm 13: FastAnchorWords [3]

```
Input: Empirical word co-occurrence matrix $\tilde{Q} \in \mathbb{R}^{n\times n}$ whose rows are almost in a
  simplex, and $\epsilon > 0$
Output: An index set $\mathcal{K}$ of $r$ row indices of rows of $\tilde{Q}$ which are close to the
  vertices of the simplex
  Project the columns of $Q^T$ onto a random $n$-dimensional subspace of $\mathbb{R}^{4\log(n)/\epsilon^2}$
  using the Fast Johnson-Lindenstrauss Transform (FLJT)[b], let $R$ be the matrix
  resulting from the projection
  $\mathcal{K} = \{\}$
  for $i = 1$ to $r$ do
      $q = \arg\max_j \|R_{:,j}\|_2$
      $R = \big(I - \frac{R_{:,q}R_{:,q}^T}{\|R_{:,q}\|_2^2}\big)R$
      $\mathcal{K} = \mathcal{K} \cup q$
  end for
  for $i = 1$ to $r$ do
      Let $k_i$ be the $i$-th element of $\mathcal{K}$
      Compute the projection of $R$ onto the orthogonal complement of
      $R(\mathcal{K} \setminus \{k_i\}, :)$:
          $R' = (I - DR(\mathcal{K} \setminus \{k_i\}, :)R(\mathcal{K} \setminus \{k_i\}, :)^T)R$      (7.33)
      $q = \arg\max_j \|R'_{:,j}\|_2$
      $\mathcal{K} = \mathcal{K} \cup \{q\} \setminus k_i$
  end for
  return $\mathcal{K}$
```

[b]Arora et al. use the implementation of this projection given in [1], which results in a runtime of $O(n^2 + n\log(n)/\epsilon^2)$

Furthermore, the algorithm comes with provable robustness to noise. In order to formalize this robustness we must some introduce definitions. The first allows us to quantify how distinct each of the topics, or vertices of the convex hull, are from each other [3]:

Definition 6. *A simplex P is γ-**robust** if for every vertex v of P, the Euclidean distance between v and the affine hull of the rest of the vertices is at least γ [3].*

This definition allows us to formalize the intuition that more distinct the topics are, the easier it should be to recover them despite noise. Nevertheless, in the presence of noise we can only hope to recover points which are "close" to the vertices of the simplex, defined as follows [3]:

Definition 7. *Let $q_1, ..., q_n$ be a set of points whose convex hull P is a simplex with vertices $v_1, ..., v_r$. Then a_i ϵ-**covers** v_j if, whenever a_i is written as a convex combination of the vertices, $a_i = \sum_{k=1}^{r} c_k v_k$, then $c_j \geq 1 - \epsilon$. Moreover, we say that a set of r points ϵ-**covers** a the vertices if every vertex is ϵ-covered by some point in the set [3].*

In the near-separable NMF problem, we can assume that the columns of W are affinely independent, so they form a simplex, making the definitions above relevant. We would like to find a set of r points that ϵ-cover the true convex hull vertices, for some small ϵ. The authors provide a theorem guaranteeing that their algorithms does just this under certain conditions:

Theorem 8. *[3] Suppose there exists a set of points $\mathcal{Q} = \{q_1, ..., q_n\}$ whose convex hull P is γ-robust and has vertices $v_1, ..., v_r$ (which are in $\mathcal{Q}$). Given a perturbation of points in $\mathcal{Q}$, namely $\{q'_1, ..., q'_n\}$, such that for each i, $\|q_i - q'_i\| < \epsilon$, and $\epsilon < \gamma^3/(20r)$, FastAnchorWords outputs a subset of $\{q'_1, ..., q'_n\}$ (or the indices of such vectors) of size r that $O(\epsilon/\gamma)$-covers the vertices $\{q_1, ..., q_n\}$.*

Notice that the norm of the noise is unspecified, so the guarantees should hold for perturbations bounded by epsilon in any norm. The proof relies on two lemmas that show that the extracted vectors $O(\epsilon/\gamma^2)$-cover the true vertices after the first set of projections and that the post-processing stage reduces the error to $O(\epsilon/\gamma)$. The proofs are given in the supplement to [3], which we not cover here.

To recover H, Arora et al. propose two algorithms, RecoverKL and RecoverL2, which try to minimize the error in terms of the K-L divergence and ℓ_2 norm, respectively [3]. In the probabilistic topic modelling framework, $H(i, j)$ is the conditional probability that the word j is generated given its topic is i. Let $\bar{Q}$ be the row-normalized version of Q. Using Bayes' rule, one can show that there is an intermediary matrix $C \in \mathbb{R}^{n \times r}$ such that $C(i, j)$ is the conditional probability that topic j corresponds to word i given word i, and $C(i, :)\bar{Q}(\mathcal{K}, :)$ should be close to $\bar{Q}(i, :)$ for all i if all the words are to be represented as convex combinations of the anchor words [3]. This motivates the objective functions in the RecoverKL and RecoverL2, which can be minimized subject to the convex constraints using the exponentiated gradient algorithm [3]. Arora et al. show in their supplementary material that given polynomially many documents, these algorithms return an estimate of the true topic-word matrix H that has additive error at most ϵ [3].

Algorithm 14: Recover L2 [3]

```
Input: Empirical word co-occurrence matrix $\tilde{Q} \in \mathbb{R}^{n \times n}$ whose rows are almost in a
   simplex, and index set of anchor words $\mathcal{K}$, tolerance parameter $\epsilon$
Output: Topic-by-word matrix $H \in \mathbb{R}^{r \times n}_{\geq 0}$
  Normalize the rows of $Q$ to form $\bar{Q}$
  Store the normalization constants $z_w$ s.t. $Q = \text{diag}(z_w)\bar{Q}$
  $\bar{Q}_{\mathcal{K}_i}$ corresponds to the $i$-th anchor word
  for $i = 1$ to $r$ do
      Solve $C(i,:) = \arg\min_{C(i,:)} \|\bar{Q}(i,:) - C(i,:)^T \bar{Q}(\mathcal{K},:)\|_2$
          –      Subject to $\sum_{k=1}^{n} C(i,k) = 1$ and $C(i,k) \geq 0$
          –      With tolerance $\epsilon$
  end for
  Assign $(H^{\natural})' = C^T \text{diag}(z_w)$
  Normalize the rows of $(H^{\natural})'$ to form $H$
  return  $H$
```

However, doing so requires that we have to solve n convex programs. Although the exponentiated gradient algorithm finds an ϵ-good solution in polynomial time [3], this still requires a substantial investment of computational resources, which is one of the reasons we choose to recover H by solving the NNLS subproblem in H with BPP. Another reason is that our experiments will not test how well each algorithm recovers distributions particular to probabilistic topic modelling context, but will instead test how well each algorithm minimizes the objective function $\|X - WH\|_F$, in which case solving $\min_{H \geq 0} \|X - WH\|_F$ is the optimal strategy given W. Choosing to solve the NNLS subproblem also makes sense for evaluating how well this algorithm solves the near-separable NMF problem, since the objective of the near-separable NMF problem is to recover W, then H is computed by solving the NNLS subproblem. This is how we generate H in SPA, so for comparisons of each algorithm's ability to recover W, it makes sense to solve for H in the same manner here. Nevertheless, we detail the RecoverL2 algorithm in 14 for reference (RecoverKL is equivalent but with the KL-diverence objective function) [3].

7.2.3 Hottopixx and LP

We now switch gears from discussing SPA-based algorithms to linear programming algorithms for solving near-separable NMF. Near-separable NMF was first formulated as a linear program by Bittorf et al. in 2012 [12], then Gillis and Luce improved on the formulation in 2014 [40]. The algorithm to solve the linear program due to Bittorf et al. is known as Hottopixx, and we refer to the algorithm to solve the linear program due to Gillis and Luce as LP.

For linear programming the near-separable NMF problem is formulated in a slightly different way. First consider that if a matrix X is r-separable, there exists a matrix $Y \in \mathbb{R}^{n\times n}$ such that

$$X = XY \text{ and } Y = \Pi^T \begin{bmatrix} I_r & H^{\natural\prime} \\ 0 & 0 \end{bmatrix} \Pi \tag{7.34}$$

where Π is a permutation matrix [3]. Y has $n-r$ rows of zeros, so the corresponding columns in X are not used to reconstruct any column in X. Rather, only the r columns of X whose corresponding rows in Y are nonzero are necessary to reconstruct all the columns of X. Thus, W is equivalent to the submatrix of X composed of the columns which are necessary to reconstruct the rest, while H is equivalent to the submatrix of Y composed of the nonzero rows. Note that the identity matrix appears in Y, so the columns of W must appear in X, consistent with the separable NMF assumption [37].

Assuming r is unknown, this leads naturally to the following optimization problem to compute a Y that is maximally row sparse [37]:

$$\min_{Y\in\mathbb{R}^{n\times n}_{\geq 0}} \|Y\|_{\text{row},0} \text{ such that } X = XY \tag{7.35}$$

The notation $\|Y\|_{\text{row},0}$ denotes the number of rows in Y that are not equal to the zero vector. Solving this problem amounts to finding the minimum r-such that X is r-separable, and finding the corresponding Y which reconstructs X from r of its columns. In the noisy, near-separable case, we can replace the equality constraint with $\|\tilde{X} - \tilde{X}Y\|_1 \leq \delta$. Moreover, the nonconvex objective function in 7.35 can be replaced with an ℓ_1-norm relaxation [37]:

$$\|Y\|_{q,1} = \sum_{j=1}^{n} \|Y\|_q \tag{7.36}$$

where $q > 1$ so that the new objective function is convex. The optimal solution to the new convex optimization problem is equivalent to the optimal solution of 7.35 as long as X is sufficiently incoherent [37]. However, for separable and near-separable NMF, we expect the columns of X to be correlated, so the incoherence requirements may not be satisfied. In particular, it is unclear how this model handles duplicates among the columns of X, and how provably robust it is to noise [37].

Bittorf et al. developed a different approach to approximately solve 7.35 [12]. For a nonnegative, near-separable $\tilde{X} = X + N$ where $X = W^{\natural}[I_r, H^{\natural'}]\Pi$ and the columns of X and $\tilde{X}$ sum to one (if the columns of $\tilde{X}$ do not, they can be normalized to do so without loss of generality), they propose to solve the optimization problem

$$\begin{aligned} \min_{Y \in \mathbb{R}^{n \times n}} \; & q^T \operatorname{diag}(Y) && (7.37)\\ \text{s. t. } & \|\tilde{X} - \tilde{X}Y\|_{\infty,1} \leq 2\epsilon && (7.38)\\ & \operatorname{Tr}(Y) = r && (7.39)\\ & 0 \leq Y(i,j) \leq Y(i,i) \leq 1 \quad \forall i,j && (7.40) \end{aligned}$$

where q is any vector in $\mathbb{R}^n$ with distinct entries [12, 37]. The constraints that the off-diagonal entries of Y are less than the diagonal entry in their row and that Y must be nonnegative ensure that if $\operatorname{diag}(Y)$ is sparse, Y is row sparse. Since the objective function tries to maximize the row sparsity of $\operatorname{diag}(Y)$, 7.37 forces Y to be row sparse. The Hottopixx algorithm solves 7.37, then sets the index set $\mathcal{K}$ equal to the set of indices of the rows of Y with the r largest diagonal entries, and $W = \tilde{X}(:, \mathcal{K})$. For its last step it computes H as $\arg\min_{H'} \|\tilde{X} - WH'\|_{(\infty,1)}$ [12]. This yields an NMF with error $\|X - WH\|_{\infty,1} \leq 2\epsilon$, provided that $\epsilon = \max_j \|N(:,j)\|_1 \leq O(\alpha(W)^2/r)$, where

$$\alpha(W) = \min_{1 \leq j \leq r} \min_{x \in \mathbb{R}^{r-1}} \|W(:,j) - W(:,\mathcal{J}_j)x\|_1, \quad \text{in which } \mathcal{J}_j = \{1,2,...,r\} \setminus \{j\} \quad (7.41)$$

is a parameter that measures how far the vertices of the convex hull of the columns of X are from the convex hull of the other vertices, which in turn measures how sensitive the data is to noise (the larger $\alpha(W)$, the less sensitive the data) [12, 40].

Gillis and Luce later modified the Hottopixx model to create the new optimization problem [40]:

$$\min_{Y \in \mathbb{R}^{n\times n}} q^T \operatorname{diag}(Y) \tag{7.42}$$

$$\text{s. t. } \|\tilde{X} - \tilde{X}Y\|_{\infty,1} \leq \rho\epsilon \tag{7.43}$$

$$0 \leq Y(i,j) \leq Y(i,i) \leq 1 \quad \forall i,j \tag{7.44}$$

where q is any vector in $\mathbb{R}^n$ with all distinct positive entries. The only differences are that the constant 2 is replaced by the parameter ρ, the trace constraint is removed, and q must have strictly positive entries. The use of ρ instead of the constant 2 expresses the flexibility of the model, since changing ρ allows imprecise knowledge of ϵ. Moreover, experimental results in [40] suggest that $\rho = 1$, not $\rho = 2$, is the optimal choice. The motivation to remove the trace constraint and forcing p to be positive is to reduce the possibilities for inappropriate weights to be put on some diagonal entries of Y [40]. Like Hottopixx, LP solves for H by minimizing the same $\ell_{(\infty,1)}$-norm error function [40].

These changes allow for a number of improvements over Hottopixx. First, the factorization rank r can be reasonably chosen by the algorithm, as Gillis and Luce specify that after solving the linear program for Y their LP algorithm sets the index set $\mathcal{K}$ equal to the set of all the row indices of diagonal entries Y greater than $1 - \frac{\min(1,\rho)}{2}$ [40]. Second, although the columns of X must still sum to one, there is no such requirement on the columns of $\tilde{X}$, so normalization is unnecessary and possible distortions of the data are avoided [40]. Finally, LP has better error bounds than those provided by Bittorf et al. For any $\epsilon = \max_j \|N(:,j)\|_1 \leq O(\alpha(W^\natural)^2/r)$, the algorithm can be used to identify columns of W with ℓ_1 error proportional to $O(\frac{r\epsilon}{\alpha(W^\natural)})$. Specifically, this error bound means that the identified W satisfies $\|W^\natural - W\Pi\|_1 \leq O(\frac{r\epsilon}{\alpha(W^\natural)})$, where Π is some permutation matrix that permutes the columns of W. Moreover, Gillis and Luce show that this error bound is order-wise optimal [40].

Unfortunately, both the Hottopixx and LP algorithms suffer from the fact that they must solve a linear program in n^2 variables. This takes $\Omega(pn^2)$ operations [40], which is impractical for large n. Gillis and Luce implement and test their algorithm using a professional optimization software package [40], and Bittorf et al. execute theirs in a distributed setup for large datasets [12]. In order for a fair comparison of the linear programming strategy with the other NMF algorithms, we implement the Hottopixx algorithm in MATLAB using the dual decomposition and incremental gradient descent solver provided by Bittorf et al. [12]. This technique alternates between minimizing the Lagrangian of 7.37 over the constraint set using projected incremental gradient descent, then taking a subgradient step with respect to the dual variables [12].

We detail the full procedure in Algorithm 15. In the algorithm outline, the vector $\mu \in \mathbb{R}^n$ contains the number of nonzero elements of each column divided by p, and the projection onto the constraint set of 7.37 is implemented as in [12]. Note that while Hottopixx sets H equal to the minimizer of the $\ell_{(\infty,1)}$-norm error, in our implementation we set it equal to the solution to the NNLS subproblem $\arg\min_{H'} \|\tilde{X} - WH'\|_F$, which uses the Frobenius norm.

Algorithm 15: Hottopixx [12]

```
Input: Nonnegative matrix $\tilde{X} \in \mathbb{R}^{p\times n}$ in which the entries of each column sum to
  at most 1; factorization rank $r$; primal and dual step sizes $s_p$, $s_d$
Output: Nonnegative matrices $W \in \mathbb{R}^{p\times r}$ and $H \in \mathbb{R}^{r\times n}$ such that $\tilde{X} \approx WH$.
  Choose $q \in \mathbb{R}^n$ with distinct entries
  Initialize $C = \mathbf{0} \in \mathbb{R}^{n\times n}$, $\beta = 0$
  for $t = 1$ to $N_{epochs}$ do
      for $i = 1$ to $p$ do
          Choose $k$ uniformly at random from $[p]$
          $C \leftarrow C + s_p \cdot \text{sign}\left[\tilde{X}(k,:)^T - C\tilde{X}(k,:)^T\right]\tilde{X}(k,:) - s_p \text{diag}(\mu \otimes (\beta\mathbf{1} - q))$
      end for
      Project $C$ onto the constraint set of 7.37
      $\beta \leftarrow \beta + s_d(\text{Tr}(C) - r)$
  end for
  Let $\mathcal{K} = \{i : C_{ii} = 1$, set $W = \tilde{X}(:,\mathcal{K})$
  Set $H = \arg\min_{H' \in \mathbb{R}^r} \|\tilde{X} - WH'\|_{\infty,1}$
  return $W, H$
```

7.3 TSVDNMF

The most serious issues with the provably-accurate algorithms discussed in the last section is that they all assume that the data is near-separable, which may be restrictive, and may involve solving linear or convex programs, which are hard to scale. The final two, and most recent, algorithms we cover make more general assumptions on the data than near-separability and do not require solving linear or convex programs, though in general they provide less robust theoretical guarantees. The first, TSVDNMF, is an SVD-and-clustering-based algorithm introduced by Bhattacharya et al. in 2016 [11]. Its advantages are that it solves the NMF problem under heavy noise with error guarantees matching those of the previous algorithms, improves on the previous best runtime, and makes weaker assumptions about the ground truth basis matrix $W^\natural$ than the previous algorithms. Furthermore, its heavy noise model subsumes other noise models assumed by previous algorithms, such as Gaussian noise [11].

Using a more general model than near-separability, TSVDNMF takes X and r as input and attempts to find nonnegative $(W, H) \approx (W^\natural, H^\natural)$, where $\tilde{X} = W^\natural H^\natural + N$, $\tilde{X} \in \mathbb{R}^{p\times n}$, $W^\natural \in \mathbb{R}^{p\times r}$, $H^\natural \in \mathbb{R}^{r\times n}$, and N is a noise matrix. Bhattacharya et al. claim that the separability-based algorithms require that the ℓ_1 norm of each column of the noise matrix is less than the ℓ_1 norm of the corresponding column of $W^\natural H^\natural$, otherwise the algorithms may select an incorrect subset of the columns of X to approximate $W^\natural$ [11]. However, Bhattacharya et al. observe that in many important applications, including topic modelling, $\|n_j\|_1 \approx \|(W^\natural H^\natural)\|_1$, contradicting the assumption. They propose a more realistic noise model, the *heavy noise* model, which requires:

$$\forall T \in [n] \text{ with } |T| > \epsilon_4 n\ ,\ \frac{1}{|T|}\Big\|\sum_{j\in T} n_j\Big\|_1 < \epsilon_4^2 \tag{7.45}$$

where ϵ_4 is some constant $\ll 1$ [11]. In words, this model means that the noise must be small when averaged over $\Omega(\epsilon n)$ columns, not necessarily that the noise in each column must be small, as required by the prior models. Indeed, the authors show that the heavy noise model subsumes other noise models, including independent Gaussian noise, general correlated noise, multinomial noise, and adversarial noise [11].

The authors make the preliminary assumption that each column of $W^\natural$ sums to 1 and each column of $H^\natural$ sums to at most 1. This can be assumed without loss of generality because we can divide each column i of $W^\natural$ by its ℓ_1 length to ensure $\|W^\natural_{:,i}\|_1 = 1$, then multiply the corresponding row of $H^\natural$ to preserve $W^\natural H^\natural$. Then we can scale each column j of $X, H^\natural$, and N equally to ensure that $\|H^\natural_{:,j}\|_1 \leq 1$, which does not affect the relative error [11].

Recall that a matrix $X = W^\natural H^\natural$ is separable if and only if after some permutation of the columns of $H^\natural$, the first r columns of $H^\natural$ form a diagonal matrix. Let this matrix be called D_0. The TSVDNMF algorithm makes a weaker assumption than separability by assuming only that the off-diagonal elements are smaller than the diagonal elements in D_0, but they are not necessarily zero. Furthermore, instead of assuming that for each row l of $H^\natural$, there is only one column $i(l)$ such that $H^\natural_{l,i(l)}$ is the only nonzero element of $H^\natural_{:,i(l)}$, the algorithm assumes that there is a set of columns S_l which have a dominant element. Specifically, there must exist r disjoint sets $S_1, S_2, ..., S_r \in [n]$ that satisfy the following requirements [11]:

1. $\forall i \in S_l$, $\forall l' \neq l$, $H^\natural_{l'i} \leq H^\natural_{li}$
2. $\forall l, \sum_{i \in S_l} H^\natural_{li} \geq p_0$
3. $\forall l$, $\forall i \in S_l$, $H^\natural_{li} > \gamma$

where ρ, p_0 and γ are constants that lie in $(0,1)$. This assumption amounts to assuming multiple 'catchwords' for each topic in the context of topic modelling, instead of one anchor word. The authors term this assumption the *Dominant Features Assumption* [11]. Here, 'features' refers to the rows of X.

Bhattacharya et al. make two additional assumptions, this time on the matrix $H^\natural$. The first is the *Dominant Basis Vectors Assumption*, which says that there exists a partition $T_1, T_2, ..., T_r$ of $[n]$ satisfying:

1. $\forall l, \forall j \in T_l, l' \neq l, W^\natural_{jl} \geq \alpha$ and $W^\natural_{jl'} \leq \beta$
2. $\forall l, |T_l| \geq b_0 n$

where α, β and b_0 are constants that lie in $(0,1)$, and $\alpha > \beta$ [11]. This assumption can be roughly summarized in the context of topic modelling as the assumption that each document has a dominant topic, and each topic dominates at least some minimum number of documents. As before, 'basis vectors' refers to the columns of W, or the word profiles of each topic for topic modelling [11].

The final assumption is the *Nearly Pure Records Assumption*: $\forall l$, there exists a set P_l of $\epsilon_0 n$ columns j of $H^\natural$ such that $H^\natural_{lj} \geq 1 - \epsilon_4$. Again in the context of topic modelling, this assumption assumes that each topic has at least some minimum weight in at least some minimum number of documents [11].

These latter three assumptions together form the *Dominant NMF* model. As we will soon investigate in greater detail, the authors show that if $(W^\natural, H^\natural)$ satisfy the conditions of the Dominant NMF model, then the TSVDNMF algorithm provably recovers them under heavy noise [11].

We are now prepared to discuss the TSVDNMF algorithm itself. The algorithm first executes a thresholding step on X (outlined in Alg. 17) then performs a truncated SVD on the thresholded matrix D, then identifies the dominant basis vectors for each data point and the dominant features (rows) for each basis vector (column of $W^\natural$), then finally finds the basis vectors themselves. After the estimate W of $W^\natural$ has been computed, H can be found by solving $\arg\min_{H \in \mathbb{R}^{p \times n}_{\geq 0}} \|\tilde{X} - W^\natural H\|_F$; here we use the block principle pivoting method to solve this nonnegative least squares problem. The full algorithm is detailed in Algorithm 16.

```
Algorithm 16: TSVDNMF [11]
  Input: Matrix $\tilde{X} \in \mathbb{R}^{p\times n}$ where $\tilde{X} = W^\natural H^\natural + N$, factorization rank $r$, other
    parameters $\alpha, \epsilon_0, \nu$
  Output: Nonnegative matrices $W \in \mathbb{R}^{p\times r}$ and $H \in \mathbb{R}^{r\times n}$ such that $W \approx W^\natural$ and
    $H \approx H^\natural$
      1. Apply Thresholding procedure (see Alg. 17) to $\tilde{X}$ to get $D$
      2. Compute $D^{(r)}$, the rank-$r$ truncated SVD of $D$
      3. Identify dominant basis vectors for each record:
          (a) Perform $k$-means clustering of columns $D^{(r)}$ with $k = r$
          (b) Apply Lloyd's algorithm to the columns of $D$, using the output of
              Step 3(a) as initialization
          (c) Let $R_1, R_2, ..., R_k$ be the $k$-partition of $n$ after Step 3(b).
      4. Identify dominant features for each basis vector:
          (a) For each $i, l$, compute $g(i,l) =$ the $(\lfloor \epsilon_0 n/2 \rfloor)$-th largest element of
              $\{X_{ij} : j \in R_l\}$
          (b) For each $l$, compute $J_l = \{i : g(i,l) > \max(\gamma - 2\epsilon_4, \max_{l'\neq l} \nu g(i,l'))\}$
      5. Find basis vectors:
          (a) For each $l$, find the $\lfloor \epsilon_0 n/4 \rfloor$ largest $\sum_{i\in J_l} X_{ij}$ among all $j \in [n]$ and set
              $W_{:,l}$ equal to the average of these $X_{:,j}$
      6. Solve $\min_{H\succeq 0} \|X - WH\|_F$ using any of the NNLS algorithms, set $H$ equal
         to the solution
    return $W, H$
```

TSVDNMF relies on many parameters. For ϵ_4 sufficiently small, its particular value does not impact the recovery of $(W^\natural, H^\natural)$ The algorithm's behavior can be completely determined by the choice of only α, ϵ_0, and ν. In our experiments, we fix $\epsilon_4 = 10^{-6}, \gamma = 2\epsilon_4$, $\epsilon_0 = 0.04$, $\alpha = 0.9$, as in [11], and $\nu = 1.05$.

Algorithm 17: Thresholding [11]

```
Input: Matrix $X \in \mathbb{R}^{p\times n}$ where $X = W^{\natural}H^{\natural} + N$
Output: Thresholded matrix $D \in \mathbb{R}^{p\times n}$
  Initialize $R := [p]$, where $R$ is the set of unpruned features
  for all $i = 1, 2, ..., p$ do
      Compute $\nu_i$, the $(1 - \frac{\epsilon_0}{2})$-fractile row $i$ of $A$. Let $\zeta_i := \alpha\nu_i - 2\epsilon_4$, where $\zeta_i$ is
      the threshold for row $i$ of $X$
      if $\zeta_i \geq 0$ then
          Set $C_i := \{j : X_{ij} > \zeta_i\}$
          for all $j = 1, 2, ..., k$ do
              if $j \in C_i$ then
                  Set $D_{ij} := \sqrt{\zeta_i}$
              else
                  Set $D_{ij} := 0$
              end if
          end for
      else
          Set $C_i := \emptyset$
          for all $j = 1, 2, ..., k$ do
              Set $D_{ij} := 0$
          end for
      end if
  end for
  Sort the $|C_i|$ in ascending order, re-number the indices $i$ so that the $|C_i|$ are in the
  ascending order {Pruning Step}
  for all $i \in R$ do
      for all $i' \in R$ AND $i' > i$ do
          if $C_i \tilde{\subseteq} C_{i'}$[c] then
              for all $j \in C_{i'} \setminus C_i$ do
                  Set $D_{i'j} = 0$
              end for
              Remove $i'$ from $R$
          end if
      end for
  end for
  return $D$
```

[c]If $C_i, C_i' \subseteq [n]$, the authors write $C_i \tilde{\subseteq} C_{i'}$ to denote $|C_i \setminus C_{i'}| \leq \epsilon_0 n/4$, meaning that there are only a small number of elements in C_i that are not also in $C_{i'}$ [11]

The motivation for the thresholding procedure is to select the elements of X which are in $\cup_{l=1}^{r} S_l \times T_l$, i.e. elements which formed by the product of corresponding dominant features and dominant basis vectors. In the context of topic modelling, this means that the thresholded elements should correspond to catchwords for a particular topic and the documents for which that particular topic is dominant. Thus, we would like to compute a block diagonal D. However, with noise the thresholding procedure only approximately satisfies the above. Moreover, not necessarily all of the features are dominant for some basis vector, which complicates the thresholding. The careful selection of each row threshold and the pruning of the rows of D help to cause those features which which are not dominant for any basis vector to satisfy block diagonality in D [11].

Next, the rank-r SVD of D yields a $D^{(r)}$ whose columns are projected onto the top r singular factors of the data, which allows for a natural clustering into r clusters by the k-means algorithm. This clustering is a sensible initialization for Lloyd's algorithm to cluster the columns of D. Now, we can assume that the points in each cluster share a dominant basis vector, with that basis vector being the extreme point of the cluster in relation to the other data points (since the basis is theoretically the best basis of cardinality r to represent the data, it must include all of the extreme points, from which the interior points can be represented - recall that the columns of W can be imagined as the vertices of the convex hull of the data in an ideal scenario). However, since the basis vectors are the extreme points of the clusters, the cluster members cannot simply be averaged to find the basis vectors. Instead, the algorithm attempts to average only those members which are close to the extreme point. To do so, it finds the dominant features among each cluster (Step 4), then finds the data points which have the largest amounts of these features, then computes the average of these data points as the approximation to the basis vector corresponding to that cluster [11].

Surprisingly, Bhattacharya et al. give no proof that the W output by TSVDNMF is nonnegative, nor do their proofs of the accuracy of the algorithm obviously imply that W is nonnegative [11]. The elements of W appear to be able to be negative because each column is computed as the average of some set of columns of $\tilde{X}$, which may contain negative elements due to the noise matrix N and the lenient noise model. Perhaps, then, an additional assumption must be made on $\tilde{X}$, i.e. that it must be nonnegative [11].

Conversely, the authors do provide provable bounds on the error and runtime of TSVDNMF, as well as show an important identifiability result. Their theorem bounding the error states:

Theorem 9. *[11] Under the heavy noise and dominant NMF assumptions, for each column l, the matrix W returned by TSVDNMF satisfies:*

$$\|W_{:l} - W^{\natural}_{:l}\|_1 \le \epsilon_0 \tag{7.46}$$

This error guarantee matches those of previous algorithms [11], and is important because it is achieved via weaker assumptions and a smaller runtime (see below).

Meanwhile, the authors' identifiability result is unique in that it does not assume the separability of $W^{\natural}H^{\natural}$ [28] nor use hard-to-verify geometric conditions [37]. Recall that a solution is identifiable if it is the only solution that achieves the minimal error. Here the authors show when a result is *approximately identifiable*, a term which they define as:

Definition 10. *[11] Given r and a matrix $X \in \mathbb{R}^{(p\times n}$, the matrix $W^{\in \mathbb{R}^{(p\times n)}}$ is* ***approximately identifiable*** *from X if there exists an H such that $\|X - WH\|_1 \le \epsilon$ and for all $W'H^{\natural'}$ where $\|X - W'H^{\natural'}\|_1 \le \epsilon$, we have that for some column permutation Π, $\|W - \Pi(W')\| \le \epsilon'$ and $\epsilon' \to 0$ as $\epsilon \to 0$.*

This means that if the error of any estimate $(W', H^{\natural'})$ goes to zero, then W' must converge to W, making W the unique optimal solution [11].

The authors' approximate identifiability theorem assumes that the ground truth basis matrix $W^{\natural}$ satisfies the Dominant Features Assumption and its columns sum to one and the ground truth weight matrix $H^{\natural}$ satisfies the Nearly Pure Records Assumption [11]. Then, at a high level, if any estimate (W, H) has a sufficiently small error and multiple other conditions are satisfied, W must be close to $W^{\natural}$ up to permutation and scaling of the columns. The sufficiently small error is determined by the ℓ_1 norm of the cumulative difference between the column sums of $W^{\natural}H^{\natural}$ and WH in each cluster yielded by Step 3 of the TSVDNMF algorithm of being smaller than the scaled sum of the ℓ_1 norms of particular columns of $H^{\natural}$ [11]. Namely, the following must hold for all l:

$$\Big\| \sum_{j\in R_l} (W^{\natural}H^{\natural})_{:,j} - \sum_{j\in R_l} (WH)_{:,j} \Big\|_1 \le \delta \sum_{j\in P_l} \|H^{\natural}_{:,j}\|_1 \tag{7.47}$$

Furthermore, defining $\delta' := 2\epsilon_4 + 6\delta$, the following conditions must also hold:

1. $(p_0 - \delta')^2 > 4k(p_1 + \delta')$
2. $2\delta' < p_1 - p_0$

where p_0 is the constant used in the Dominant Features Assumption and p_1 is a constant that is not specified elsewhere - apparently it is unconstrained except by the above two conditions, so as long as there exists some value for p_1 which satisfies the above, p_1 can take on that value [11]. Then, there exists a permutation $\pi : [r] \to [r]$ and constants $\alpha_j \in \mathbb{R}$ for $j \in [k]$ such that for all j:

$$\|W^{\natural}_{:,j} - \alpha_j W_{:,\pi(j)}\|_1 \leq 2\delta' + \frac{4k(p_1 + \delta')}{p_0 - \delta'} \tag{7.48}$$

We refer the reader to the supplement of [11] for the proof. This result shows roughly that the optimal NMF of a matrix that satisfies certain assumptions is unique, but the importance of this theorem is questionable. First, it does not show that "a uniqueness result holds even for unrestricted NMF...making it well-posed" as the authors claim elsewhere in the paper [11], since the uniqueness result only holds for the NMF of a matrix that satisfies the Dominant Features Assumption and the Nearly Pure Records Assumption. In fact, the unrestricted NMF problem has been shown to be ill-posed [49], so any result claiming to show it is well-posed should be treated with skepticism. Furthermore, its usefulness even for solving the restricted NMF problem (restricted by the noted assumptions) is dubious. The terms $W^{\natural}H^{\natural}$, $H^{\natural}$, p_0, and P_l are unknown in practice, so the practitioner has no way of knowing whether the NMF problem satisfies the model assumptions. Thus, we recommend further research into more useful identifiability tests for restricted NMF problems.

Bhattacharya et al. also state that the overall runtime of the algorithm is $O((p+n)^2r)$, which improves upon the previous best runtime of $O(\max(p,n)^3r)$ [11]. However, we would like more explanation for how they determined this complexity. According to our analysis, executing the k-means algorithm and Lloyd's algorithm requires $O(prn)$ operations, assuming that the number of iterations is upper bounded by a constant (it is bounded by 50 in our implementation). Similarly, Step 4(a) can be executed in $O(prn)$ iterations to iterate over n elements for every (i,l), and Step 4(b) can be executed in $O(pr)$ operations to select the maximizing $l \in [r]$ for every $i \in [p]$. The basis vector estimates in Step 5 can be computed in $O(prn)$ operations, since for each l, $O(n)$ elements must be examined, and $O(n)$ vectors of length p must be added. As we discussed in Section 2.3, Step 6 can be solved in $O(prn)$ operations. This leaves the thresholding procedure and the rank-r truncated SVD as the remaining unaccounted-for costs. The first for-loop of the thresholding procedure requires $O(pn)$ iterations, executing $O(n)$ iterations on each of p loop iterations. Sorting the $|C_i|$ requires $O(p \log p)$ iterations, and the authors remark that the pruning loop can be executed correctly with high probability in $O^*(p^2)$ operations using a randomized version of the step as described in Algorithm 17 (otherwise this step would take $O(p^2 n)$ operations). Yet we are still below the stated $O((p+n)^2 r)$ time cost, so this cost must be entirely due to computing the rank-r truncated SVD. Yet state-of-the-art algorithms can compute a truncated rank-r SVD of a p-by-n matrix in $O(prn)$ operations [41]. So we are unclear where the $O((p+n)^2 r)$ cost comes from, and request further clarification.

In sum, the TSVDNMF algorithm achieves error guarantees commensurate with previous algorithms while claiming to have a faster runtime and making weaker assumptions on $W^\natural$ and N, and new assumptions on $H^\natural$. Although the assumptions on $H^\natural$ are new and hence stricter than previous work (which did not make any significant assumptions on $H^\natural$), the authors examine the estimate to $H^\natural$ yielded by the PW-SPA algorithm [42] for five different real datasets and find that the assumptions are consistent with these estimates of $H^\natural$ [11]. We will supplement their empirical results with further tests of the algorithm's accuracy and runtime on synthetic dataset in Chapter 8.

Speaking of our experiments, in an effort to test the universal applicability of TSVDNMF, the data we test on does not necessarily satisfy the assumptions of dominant NMF. This means that some of the basis vectors may not have dominant features, which would causes a division by zero in Step 5 of the TSVDNMF algorithm. To minimize the likelihood of this, we lowered the dominant threshold factor ν from the recommended 1.15 to 1.05, and to account for cases where basis vectors without any dominant features remained, we modified the TSVDNMF algorithm to set any column corresponding to a basis vector without any dominant features to the average of all of the column vectors of X in its cluster.

7.4 Union of Intersections Clustering

Even though the dominant NMF assumptions are generally weaker than the separability and near-separability assumptions and the heavy noise model subsumes previous noise models, these two sets of assumptions are still similarly as hard to check in practice as the prior assumptions. This is primarily because the ground truth factors $W^\natural$ and $H^\natural$ are unknown, and the assumptions on them cannot be verified until they are found. Furthermore, the dominant NMF and heavy noise models are stringent enough that it is entirely possible for real datasets to not satisfy them, for example in many real datasets the noise distribution is unknown, and could very well violate the heavy noise assumptions. Thus, an NMF algorithm which can achieve high performance while relying on more verifiable assumptions, or even better, no assumptions at all on the data, is highly desirable. The most recent NMF algorithm we discuss lacks the error guarantees of prior algorithms, but it generates insightful and accurate factors using two novel, intuitive strategies while making no non-trivial assumptions on the data [71], making it an important contribution to the NMF literature.

This algorithm is called the Union of Intersections clustering algorithm (UoI-$NMF_{cluster}$), and was proposed by Ubaru et al. in 2017 [71]. The algorithm employs the recently introduced Union of Intersections framework [14], which separates feature learning from weight learning, uses bootstrap sampling to find stable features, and intersection and union operations to induce sparsity and reduce variance in the generated estimates. While other NMF algorithms attempt to find factors W and H that minimize the error $\|X - WH\|_F$ as the means to obtain factors that are robust to noise (stable) and interpretable, $UoI - NMF_{cluster}$ attempts to find stable and interpretable factors directly. As such, the authors provable no provable bounds on the error of the solution yielded by their algorithm, but they do provide brief experimental results in which UoI-$NMF_{cluster}$ outperforms most of the other NMF algorithms they test in terms of error. Like previous work, they model the data X as composed of ground-truth nonnegative factors $W^\natural$ and $H^\natural$ and noise N, such that $X = W^\natural H^\natural + N$ [71].

7.4.1 Stability

To achieve stability, i.e. robustness to noise, the algorithm first selects the most stable basis vectors over NMF results on many bootstrap samples from the data [71]. The intuition for the former strategy is the assumption that in each bootstrap sample (subset of the columns of X), there will be some basis vectors corresponding to de-noised parts of the data, and some which correspond to noise or are spurious. The basis vectors corresponding to different de-noised parts of the data must be very far from each other spatially, since different parts of the data presumably have very different spatial representations. Meanwhile, the data vectors corresponding to the noise will be scattered somewhere in between. Over many bootstrap samples, the computed basis vectors corresponding to the same parts of the data will be very similar to each other (stable) and distinct from those corresponding to other parts of the data, whereas the computed spurious basis vectors and those corresponding to the noise will be scattered all over the space, at least according to Ubaru et al. [4] If this reasoning is correct, the computed basis vectors will form distinctive clusters around each of the true basis vectors of the data, whereas the computed basis vectors representing noise will be scattered outside of the clusters [71].

To recover the true basis vectors, the algorithm uses the clustering algorithm Density Based Spatial Clustering of Applications with Noise (DBSCAN) [31] which computes clusters that are densely packed, and designates points outside the dense clusters as noise. Ubaru et al. set the DBSCAN parameter $MinPts$ (the minimum number of points per cluster) equal to approximately half the number of bootstrap samples, dictating that a basis vector must appear in at least approximately half of the bootstrap samples in order for it to be considered stable and made into a cluster by DBSCAN [71]. Taking a step back, to compute the NMF factorization of each bootstrap sample, $UoI\text{-}NMF_{cluster}$ uses the version of the multiplicative updates algorithm which aims to minimize the Kullback-Leibler (K-L) divergence of the error instead of the Euclidean norm. This algorithm (which we denote MU-KL) was introduced in the same paper as the MU algorithm [59] and has the same structure but with different update rules:

$$w_{ik} \leftarrow w_{ik} \frac{\sum_j h_{kj} x_{ij}/(WH)_{ij}}{\sum_l h_{kl}} \tag{7.49}$$

$$h_{kj} \leftarrow h_{kj} \frac{\sum_i w_{ik} x_{ij}/(WH)_{ij}}{\sum_l w_{lk}} \tag{7.50}$$

Drawing on the notion of stability, $UoI\text{-}NMF_{cluster}$ can find the factorization rank r which yields the most 'stable' factorization. In fact, a factorization rank selection procedure is built into the algorithm, but we do not include it in our implementation because we assume throughout this work that the factorization rank is given. To select the best r, the algorithm chooses the r that minimizes the average of the dissimilarities between the sets of basis vectors W computed across bootstrap samples:

$$\Gamma(r) = \frac{2}{B_1(B_1-1)} \sum_{1 \le i \le j \le B_1} diss(W^{(i)}, W^{(j)}) \tag{7.51}$$

[4]It is not entirely clear why there will not also be stable noise basis vectors since the underlying noise matrix does not change from sample to sample. It is also unclear how we know that the NMF algorithm will yield basis vectors that are either true basis vectors of the data or noise basis vectors, but not mixtures of both. We further analyze these assumptions later in this section.

where the dissimilarity measure $diss(W, W')$ is the one introduced in [78] and is a function of the cross correlation matrix $C = W^T W'$:

$$diss(W, W') = \frac{1}{2r}(2r - \sum_{j=1}^{r} \max_i C_{ij} - \sum_{k=1}^{r} \max_j C_{ij}) \tag{7.52}$$

7.4.2 Interpretability

The second aim of the algorithm, interpretability, is primarily achieved thru the algorithm's second stage, the computation of the weight matrix H, because the H it computes is highly sparse [71]. We have seen that sparse initial factor estimates can be detrimental to NMF algorithm performance (see Chapter 5), but as mentioned in the introduction, sparse final factors significantly aid interpretability. Sparsity in the weight matrix is particularly beneficial for interpretability because it means that each data point in X is composed of only a small number of basis vectors in W, making it easy to classify data points according to the basis vectors which compose them.

$UoI\text{-}NMF_{cluster}$ forces H to be highly sparse through the Union of Intersections procedure [14]. After computing the basis matrix W with DBSCAN, the algorithm recomputes each weight matrix from each of the original bootstrap samples by solving the nonnegative least squares (NNLS) problem with W fixed. Next, for each column j, the algorithm computes the intersection of the supports of each column of each weight matrix $H^{(i)}$ which was generated from a bootstrap sample containing the j-th column of X. It saves the result in the j-th column of the index matrix $H^{(idx)}$, yielding a highly sparse $H^{(idx)}$ after completion. Then, the algorithm repeats the bootstrapping procedure with new samples, and this time computes only the indices of the weight matrices specified in $H^{(idx)}$ by solving distributed NNLS problems. We use BPP to solve the NNLS problems in our implementation. Lastly, the algorithm computes the final weight matrix as the average (roughly, the union) over the weight matrices generated from the last series of computations on bootstrap samples. This final matrix is still sparse, and hence interpretable, because it is the scaled sum over matrices that have nonzero entries that are subsets of the nonzero entries in $H^{(idx)}$. Furthermore, the Ubaru et al. note that the solution has low variance because of the bootstrap sampling, and no bias because of no explicit regularization [71].

An outline of the algorithm is shown in 18. Here, to account for vector matrix indexing, we use MATLAB notation for matrix indexing. That is, $A(x, y)$, where x and y may be vectors, represents the submatrix of A picked out by the row indices in x and the column indices in y. Ubaru et al. suggest setting $B_1 = 20$ and $B_2 = 10$, but we obtain better results with larger B_1 and B_2, so we set them to 80 and 40, respectively.

Similarly, we choose to set k and k' both equal to $n/5$ in our implementation after experimenting with numerous values.

Algorithm 18: *UoI-NMF*$_{cluster}$ [71]

```
Input: $X \in \mathbb{R}_{\geq 0}^{p \times n}$, factorization rank $r$, numbers of bootstrap resamples $B_1, B_2$
Output: $W \in \mathbb{R}_{\geq 0}^{p \times r}$ and $H \in \mathbb{R}_{\geq 0}^{r \times n}$ such that $X \approx WH$
  1) Basis Learning and Selection
  for $i = 1$ to $B_1$ do
      Sample $k$ indices from $[n]$, call the set of these examples $s_i$
      Compute $[W^{(i)}, H^{(i)}] = \text{MU-KL}(X(:, s_i), r)$
      Append $S(i, :) = s_i$, $\tilde{W} = [\tilde{W}, W^{(i)}]$
  end for
  1a) Choose the best set of bases
  Cluster the stacked matrix $\tilde{W}$ using DBSCAN with min. $B_1/2$ points/cluster
  Set the columns of $W$ (basis vector estimates) equal to the cluster centroids
  1b) Update the weights and intersection of supports
  for $i = 1$ to $B_1$ do
      Compute $H^{(i)} = \arg\min_{H \in \mathbb{R}_{\geq 0}^{r \times p}} \|X - WH\|_F$ using BPP
  end for
  for $j = 1$ to $n$ do
      Let $T = \langle (row_1, col_1),...,(row_{q_j}, col_{q_j})\ \rangle$ be the row and column indices of each
      of the $q_j$ entries in $S$ that equal $j$.
      for $l = 1$ to $r$ do
          if $\cap_{i=1}^{q_j} (H_{l,col_i}^{(row_i)} > 0)$ then
              Set $H_{l,j}^{(idx)} = 1$
          else
              Set $H_{l,j}^{(idx)} = 0$
          end if
      end for
  end for
  2) Weight Estimation
  for $i = 1$ to $B_2$ do
      Initialize $H^{(i)} = \mathbf{0}$
      Sample $k'$ indices from $[n]$, call the set of these examples $s'_i$
      for $j = 1$ to $k'$ do
          Set $h^{(idx)} = H^{(idx)}(:, s'_i(j))$
          Let $rows$ be the vector of indices where $h^{(idx)}$ is nonzero
          Update $H^{(i)}(rows, s'_i(j)) = \arg\min_{v \geq 0} \|X(:, s'_i(j)) - W(:, rows)v\|_F$,
          solve NNLS problem using BPP
      end for
  end for
  Set $H = \frac{1}{B_2} \sum_{i=1}^{B_2} H^{(i)}$
  return $W, H$
```

DBSCAN executes in only $O(d \log d)$ iterations to cluster d points [31], so the most expensive parts of the algorithm are executing MU-KL on the p-by-k matrices $X(:, s_i)$, which takes $O(prk)$ operations per call to MU-KL (since we hold the number of iterations constant at 100) and solving the NNLS problems using BPP, which each take $O(prn)$ operations (see Section 2.3). Considering B_1, B_2, k and k' constants, the number of calls to these functions is constant, total time cost is thus $O(prn)$, equivalent to the NMF heuristics, but likely with a larger associated constant.

7.4.3 Further Analysis

We now return to the assumptions that the computed basis vectors corresponding to the noise matrix N are not stable across bootstrap samples, and that running an NMF algorithm on the bootstrap sampled data will yield basis vectors that either represent noise or the true data basis vectors. What follows is our own analysis of these assumptions.

Let $X(:, s)$ be a particular bootstrap sample of the data, where s is a set of (not necessarily unique) column indices of X, and $X(:, s)$ is the matrix composed of the columns of X specified by the indices in s. Also, let $|s|$ be the number of indices in s. We have:

$$X(:, s) = W^{\natural} H^{\natural}(:, s) + N(:, s) = [W^{\natural} \quad N(:, s)] \begin{bmatrix} H^{\natural} \\ I_{|s|} \end{bmatrix} \tag{7.53}$$

The matrix on the right-hand side has inner dimension larger than at least one of its outer dimensions, since the H has $r + |s|$ rows but only $|s|$ columns. We may solve for the factors $[W^{\natural} \quad N(:, s)], [H^{\natural} \quad I_{|s|}]^T$ using a standard NMF algorithm, but the meaningfulness of the solution generated by these methods when the factorization rank is larger than the nonnegative rank of the matrix is dubious because there may be infinite solutions, all with error approaching zero. In this case, to the best of our knowledge there is no way to force the NMF algorithm to converge to a particular unknown solution. We have considered extending $X(:, s)$ by concatenating copies of its columns to itself, so that the new matrix would have dimensions larger than the factorization rank, but this new matrix would still have nonnegative rank lower than the factorization rank. Experimentally, we observe that MU generates a different basis matrix on every execution on the same dataset with nonnegative rank less than the factorization rank. For now we assume that there exists some NMF algorithm that finds the optimal factorization outlined above, which perhaps may be achieved by fixing the bottom half of the weight matrix to be the identity matrix during the execution of the algorithm.

Now, computing the rank-$(r+|s|)$ factorization of the bootstrapped sample should yield the true basis vectors of the data and basis vectors corresponding to noise. Over many bootstrap samples, we will thus observe the computed basis vectors cluster into r clusters corresponding to the true basis vectors, or 'parts', of the data, and n clusters corresponding to each column of the noise matrix N. These latter n clusters will be just as dense as the clusters corresponding to the true vectors, because both sets of clusters correspond to bases which are stable in the data. This seems to be what is happening in the image of the clusters (Figure 1) given in the $UoI\text{-}NMF_{cluster}$ paper [71], since each of the clusters here are dense and there are very few vectors scattered between the clusters, contradicting what we would expect if the noise were unstable.

We devise a simple solution to settle this problem of computing dense factors that are actually noise. Noting that we want the computed noise vectors to vary significantly across samples, we suggest adding a small amount of random noise to each bootstrap sample, such that the noise varies between different samples of the same data points and cannot be mistaken as stable. Indeed, this added random noise will be encompassed in $N(:, s)$ for every sample, so the noise basis vectors will differ in every sample. Once the basis vectors have been computed for every bootstrap sample, the clusters corresponding to true basis vectors of the data will be much easier to identify, as the basis vectors corresponding to noise will be spread across the space in an unstable manner.

Even if the optimal rank-$(r+|s|)$ factorization cannot be identified by any NMF algorithm, it still makes sense to add noise to each bootstrap sample. Otherwise, it is unclear how, for the same column index sampled on two different occasions, the noise on the corresponding data point will be any different. If the noise does not change across samples of the same data point, then the basis vectors it yields will be stable, whereas if the noise does change, intuitively we would expect the corresponding basis vectors to be unstable.

However, the even more important concern is that it remains unproven that the nonnegative basis vectors of $X = W^{\natural}H^{\natural} + N$ are distinctly either the true data basis vectors (columns of $W^{\natural}$) or the basis vectors of the noise N. Our formulation 7.53 attempts to show that it is possible to achieve an NMF that fully distinguishes the two types of basis vectors, yet this factorization is nontrivial to obtain. The authors of the $UoI\text{-}NMF_{cluster}$ paper provide no justification for why the NMFs they compute have basis vectors that belong to strictly one of the two categories, so it is more likely that the computed basis vectors are linear combinations of noise and the true basis vectors of the data. Thus, the clusters obtained are similarly noisy as the basis vectors obtained by a simple NMF algorithm such as MU-KL acting on a noisy matrix.

Nevertheless, the experimental results shown in [71] suggest that $UoI\text{-}NMF_{cluster}$ is effective at distinguishing (and removing) noise from the underlying data basis vectors, despite the lack of mathematical justification for this behavior. We perform our own tests to determine whether the basis vectors MU-KL yields can be clearly distinguished between noise and data basis vectors. Our test is composed of four steps:

1. Generate $W^\natural \in \mathbb{R}_{\geq 0}^{p\times r}$, $H^\natural \in \mathbb{R}_{\geq 0}^{r\times n}$ and $N \in \mathbb{R}_{\geq 0}^{p\times n}$:
   (a) For all (i,j), sample $w_{ij}^\natural \sim 2|\mathrm{N}(0,1)|$
   (b) Construct $H^\natural$ as 80% sparse, to ensure a sparse weight matrix as is the case with the the datasets tested in [71]. Let the indices (i,j) of nonzero elements be distributed uniformly across $H^\natural$, and for each (i,j) corresponding to a nonzero index, sample $h_{ij}^\natural \sim |\mathrm{N}(0,1)|$
   (c) For all (i,j), sample $n_{ij} \sim$ Unif(0,1) to ensure the noise is distributed differently from the data

2. For $i = 1$ to B:
   (a) Generate k random indices uniformly sampled from $[n]$, call the set of these examples s_i
   (b) Compute $[W^{(i)}, H^{(i)}] = \text{MU-KL}(X(:,k), r)$
   (c) Append $\tilde{W} = [\tilde{W}, W^{(i)}]$

3. For $j = 1$ to rB:
   (a) Compute $q_j = \max_{i\in[r]} \dfrac{\tilde{w}_j^T w_i^\natural}{\|\tilde{w}_j\|\|w_i^\natural\|}$
   (b) Compute $m_j = \max_{i\in[n]} \dfrac{\tilde{w}_j^T n_i}{\|\tilde{w}_j\|\|n_i\|}$

4. Plot the q_j and m_j values

Here, we use $p = 40$, $r = 8$, $n = 24$, $k = |s_i| = 16$ and $B = 20$. We expect to see a threshold separating the q_j values: if a computed basis vector is roughly a true data basis vector, its normalized inner product with one of the columns of

$W^\natural$ should approach 1, whereas if the computed basis vector is a basis vector of the noise, it should not point in a similar direction as any of the columns of $W^\natural$, so it should not have a large normalized inner product with any of them. We do not need to construct the noise to be orthogonal to the data, since $UoI\text{-}NMF_{cluster}$ makes no such assumptions on the noise. For the same reason, we also expect to see a threshold separating the m_j values. If a particular q_j is close to 1, meaning the j-th computed basis vector represents a true data vector, we expect the corresponding m_j to be far from 1, since that computed basis vector should have a small noise component, and vice versa.

However, our results contradict these expectations, suggesting that each computed basis vector has significant data *and* noise basis vector components. Figure 7.1 shows no threshold among the p_j values nor the m_j values.

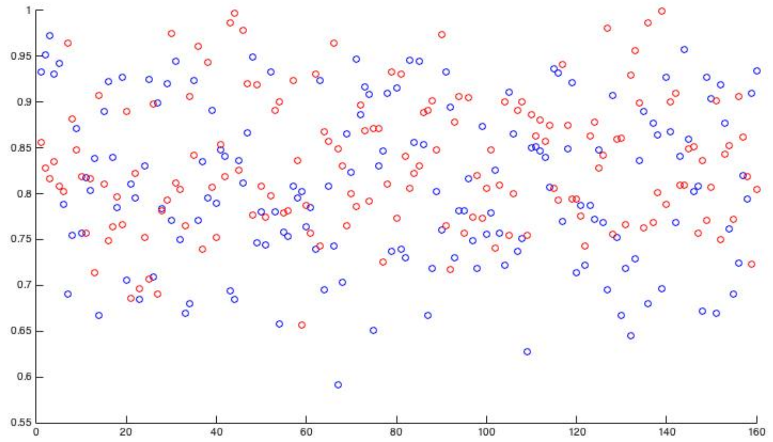


Figure 7.1: p_j (blue) and m_j (red) values when $W^\natural$ and N for the test explained in 7.4.3

Therefore, the assumption that running an NMF algorithm on bootstrap samples from X will yield basis vectors that are either true basis vectors of the data or noise basis vectors is not necessarily true.

It is perplexing, then, why $UoI\text{-}NMF_{cluster}$ yields relatively low K-L divergence error and recovers distinctive and representative features in the four experiments on noisy data presented by Ubaru et al. [71]. We suggest that it may have to do with the way three of the four datasets tested in the paper are constructed[5]. These three datasets are formed by concatenating many versions of a ground-truth matrix with different noise matrices added to it. This amounts to roughly fulfilling our

[5]The fourth dataset, the mouse brain dataset, is highly preprocessed, which also may make it more conducive to factorization by $UoI - NMF_{cluster}$ [71].

earlier suggestion that different noise be added to each bootstrap sample so that the noise is unstable. However, the authors provide no explanation that their algorithm only works on datasets of this type. Also, our preliminary testing of concatenated synthetic datasets with different noise added to them revealed that $UoI\text{-}NMF_{cluster}$ did not yield a significantly more accurate solution when it factored the concatenated matrices. Furthermore, it remains unclear that the computed basis vectors on each NMF call will be either data basis vectors or noise basis vectors even if the noise is unstable. Thus, there are still many unanswered questions about the performance of $UoI\text{-}NMF_{cluster}$.

Overall, $UoI - NMF_{cluster}$ is an innovative algorithm that brings multiple interesting new strategies to the realm of NMF. We treat it with greater analysis and skepticism than the previous NMF algorithms because it is the most recent and least studied algorithm, and because its new strategies may alter the NMF landscape if they are fleshed out with analytical rigor. Our analysis here is a step in this direction because it draws attention to the weak spots in the intuition justifying the $UoI\text{-}NMF_{cluster}$ algorithm. Despite its theoretical holes, initial tests in [71] indicate that the algorithm performs well in practice (outperforming both MU-KL and TSVD-NMF in both comparative tests in [71]), meriting further tests of the algorithm in the following section.

Chapter 8

Tests of Recent Algorithms

Additional testing of the algorithms we discussed in the previous chapter is a needed contribution to the literature because simply because they are relatively new and have yet to be thoroughly tested, especially on synthetic data of various types. Yet there are additional reasons why it is important to further tests these algorithms.

For one, although many of the recent algorithms we have discussed come with theoretical performance guarantees, these guarantees often do not paint a clear picture of how the algorithms behave in practice because they rely on assumptions that are restrictive and hard to check. When the data does not satisfy the assumptions, the algorithms' behavior is both ill-defined and ill-tested. If the data does satisfy the algorithms' assumptions, the performance guarantees on the provably correct algorithms have generally not been shown to be optimal, so their effectiveness in these cases is still questionable. In particular, the provably correct algorithms may be outperformed by the recent heuristics and/or the heuristics discussed in Chapter 2, even on separable or near-separable data. Yet tests comparing NMF heuristics to provably correct algorithms is scarce in the literature; the only such test we found was in [71].

This gap in empirical understanding of NMF algorithms motivates our tests of the recent algorithms along with MU, HALS, and ANLS BPP on varying types of synthetic data. In all of the tests, $p = 50, n = 250$, and $r = 10$ unless otherwise noted. For all algorithms which require an initialization, we use the random initialization strategy described in Chapter 5. Per usual, we measure the relative error in the Frobenius norm, plot this error over time, and execute all the tests in MATLAB. Since many of the recent algorithms calculate one estimate at the end of their execution instead of iteratively updating an estimate, we plot those algorithms' error curves as downward step functions whose step occurs at the time when the algorithm finishes executing.

8.1 Vanilla Gaussian Data

Like in Chapters 3 and 6, we execute the first test as a baseline on a dataset X where each element is the absolute value of an independent sample from the standard normal distribution. X is clearly not separable because, as we have seen previously, its nonnegative rank is much larger than r. We display the results in Figure 8.1.

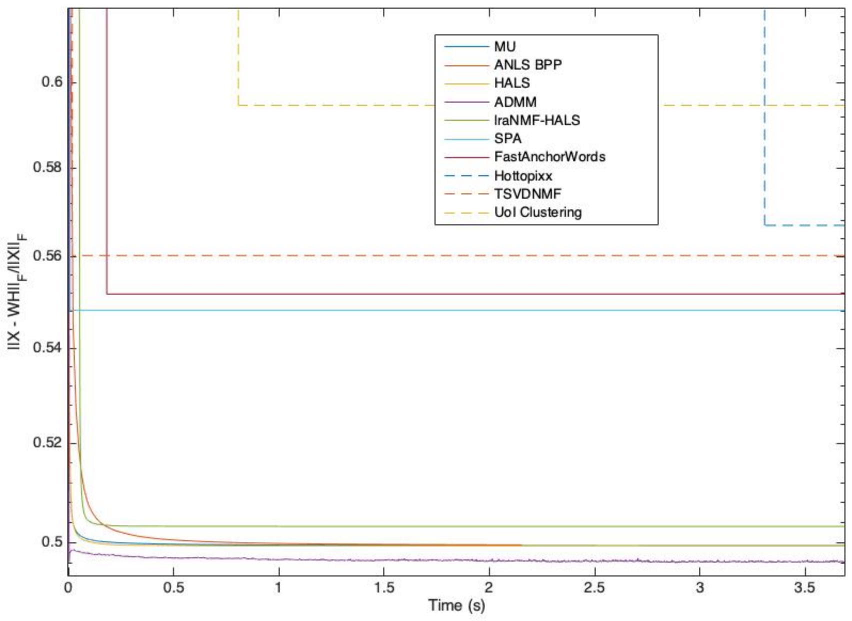


Figure 8.1: NMF relative error vs time for 10 different algorithms on the dataset X, where $X_{ij} \sim |N(0, 1)|$.

The results paint an interesting picture. First, note that ADMM is clearly the fastest algorithm to converge and the algorithm that has the lowest error at convergence. ADMM is also the uniquely best performing algorithm on many of the other tests we share in this chapter, suggesting that it should perhaps overtake HALS and ANLS BPP as the state-of-the-art algorithm for solving NMF in practice.

Meanwhile, the other new heuristic, lraNMF-HALS is not as successful. It first takes a long time to compute the low rank-approximation, then converges quickly but to a sub-optimal local minimum. This result implies that low-rank approximation techniques for NMF can quickly yield a rough solution once the low-rank approximation has been computed, but the quality of this solution is limited by the fraction of information about the full dataset contained in the low-rank approximation.

The performance of the other new algorithms is interesting as well. SPA and TSVDNMF quickly yield a relatively accurate solution, while FastAnchorWords and Hottopixx also yield an accurate solution but take significantly longer to do so, especially Hottopixx, which makes sense because it solves a linear program in n^2 variables. $UoI\text{-}NMF_{cluster}$, on the other hand, is the least accurate and the second-slowest (behind only Hottopixx). These trends are important to look out for in the remainder of the tests.

8.2 Near-Separable Data

In this section we evaluate algorithm performance on data which is near-separable, and vary the power of the noise matrix $N^{(z)}$. Recall that a matrix $\tilde{X}$ is near-separable if $\tilde{X} = W^{\natural}[I_r, H^{\natural\prime}]\Pi + N^{(z)}$, where Π is a permutation matrix and N is the noise matrix.

To generate near-separable data, we first obtain the feature matrix $W^{\natural} \in \mathcal{R}^{p\times r}$ by taking the absolute value of independent random samples from the standard normal distribution. We obtain the weight matrix $H^{\natural}$ in the same manner, then post-process it by setting r randomly selected columns to the standard basis vectors to ensure $H^{\natural} = [I_r, H^{\natural\prime}]\Pi$. Next, we compute the normalized, r-separable ground-truth matrix:

$$X = \frac{W^{\natural}H^{\natural}}{\|W^{\natural}H^{\natural}\|_F} \tag{8.1}$$

We also generate the noise matrices $N^{(z)}$ by first taking the absolute value of independent samples from the standard normal distribution. Calling this preliminary matrix N, we then scale the norm of N by the scalar z such that:

$$N^{(z)} = z\frac{N}{\|N\|_F} \tag{8.2}$$

In this way, z is a rough measure of the noise power. We finally compute the matrix $\tilde{X} = X + N$. Note that since $W^{\natural}, H^{\natural}$ and N are nonnegative, $\tilde{X}$ is also nonnegative. As the sum of an r-separable matrix and a noise matrix, $\tilde{X}$ is also r-near-separable. To evaluate each algorithm's ability to recover the ground-truth matrices $W^{\natural}$ and $H^{\natural}$, we input $\tilde{X}$ to the NMF algorithms and measure the error of their computed factors W and H relative to the ground-truth matrix X. The results of our tests with $z = 0, 1/4$, $1/2$ and 1 are shown in Figure 8.2.

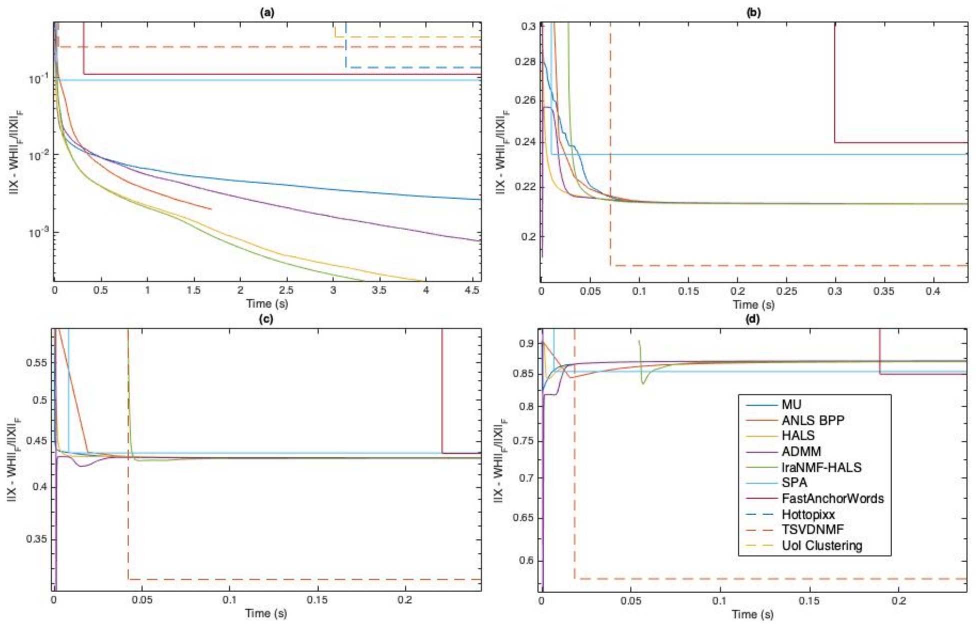


Figure 8.2: NMF relative error vs time for 10 different algorithms on near-separable matrix $\tilde{X} = X + N^{(z)}$, where $z := \frac{\|N^{(z)}\|_F}{\|X\|_F}$ and in (a) $z = 0$, (b) $z = \frac{1}{4}$, (c) $z = \frac{1}{2}$ and (d) $z = 1$. The Hottpoixx and UoI error curves are outside the scope of the plots in (b), (c) and (d). Their respective [time, rel. error] values are: (b) [3.6,0.26], [0.80,0.38] (c) [4.2,0.45], [0.76,0.52] (d) [3.1,0.85], [0.80,0.87].

Note that the only case where the iterative heuristics do not reach a stationary point within the testing window and continue to descend towards zero is the noiseless (separable) case. This is consistent with prior results in Sections 3.3: when a noiseless ground-truth solution exists, the heuristics tend to find it, and when one may not exist, they converge quickly to an erroneous local minimum. As the noise level increases, the heuristics also exhibit a greater overfitting effect, mimicking the trends in Sections 3.7 and 3.8. The overfitting effect means that the algorithms initially converge to the underlying ground-truth solution, then exhibit rapidly increasing error as they converge to the noisy solution, until they finally converge to a local minimum. This effect is especially pronounced for ADMM; for example, when $z = 1$ ADMM initially reaches a relative error of 0.31, then converges to a relative error of 0.87. Also notable is that ADMM does not perform as well as HALS, ANLS BPP, or lraNMF-HALS on the separable data, highlighting a potential weakness of this algorithm in attaining ground-truth solutions when they exist. Other than this, the performance of the iterative heuristics is very similar across noise levels.

Meanwhile, the other algorithms come close but do not exactly find the ground-truth solution on the separable data, as SPA, FastAnchorWords, Hottopixx, and

TSVDNMF all have relative errors near 0.1, and $UoI\text{-}NMF_{cluster}$ has relative error of approximately 0.33. However, these algorithms do not decline as much in performance as the iterative heuristics do when the data becomes noisy, as most of the curves converge to very similar relative errors as the data gets noisy. The one algorithm that outperforms all the others when the data is noisy is TSVDNMF, supporting that algorithm's claim to be especially robust to noise.

8.3 Sparse Data

Here we again test NMF algorithm performance over varying levels of sparsity in the data, as well as across the near-separability of the data. We design four tests: two in which the data matrix $\tilde{X}$ is near-separable, and two in which the data matrix X is not near-separable. In each of these pairs of experiments we test varying levels of sparsity.

We generate the non-near-separable, sparse matrix X by setting (100-ρ)% of the elements of X to equal the absolute value of independent random samples from the standard normal distribution, making ρ the initial percentage of sparsity in each of those factors. We test two different values of ρ: 50 and 95, and display the results in Figure 8.3.

To generate the near-separable data, we first generate $W^\natural$ and $H^\natural$ in the same manner that we generated X: we set a random selection of (100-ρ)% of their elements equal to the absolute value of independent random samples from the standard normal distribution. We then overwrite r random columns in $H^\natural$ with standard basis vectors, such that $H^\natural = [I_r, H^{\natural'}]\Pi$ for some permutation Π. Next, we generate the noise matrix N by taking the absolute values of independent random samples from the standard normal distribution. Finally, we scale $X = W^\natural H^\natural$ and N such that $\|X\|_F = 1$ and $\|N\|_F = \frac{1}{4}$, then add the two together to form $\tilde{X}$. Note that $\tilde{X}$ is not sparse because of the noise, so we are really testing how well the NMF algorithms recover the ground-truth factors when they are sparse, not when the observed matrix is sparse. We test $\rho = 50$ and $\rho = 95$, and plot the results in Figure 8.3.

The iterative heuristics are not significantly affected by the sparsity in the ground-truth factors, implying consistent noise outweighs the change in the sparsity of the factors in terms of conduciveness to factorization. On the other hand, TSVDNMF, SPA and FastAnchorWords become more accurate as the ground-truth factors become more sparse, especially FastAnchorWords, which reaches a roughly equivalent final relative error as that achieved by the iterative heuristics when $W^\natural$ and $H^\natural$ are 95%

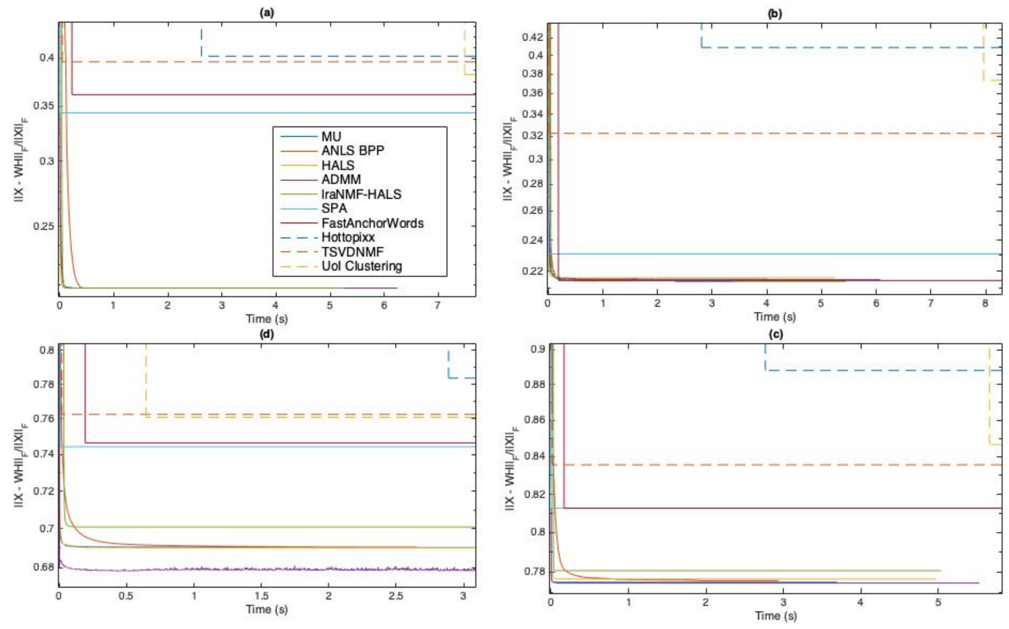


Figure 8.3: NMF relative error vs time for 10 different algorithms on $\tilde{X}$ that is (a) near-separable with 50% sparse factors; (b) near-separable with 95% sparse factors; (c) non-near-separable and 50% sparse; (d) non-near-separable and 95% sparse.

sparse. This points to the strength of these algorithms in recovering sparse factors in the near-separable setting, yet we are unsure why this is the case.

The results are very different in the non-near-separable setting. First, it makes sense that the errors in these case are much larger than in the near-separable case because with high probability any rank-r factorization of the data has very high noise. Secondly, the accuracy of each of the algorithms decreases when the sparsity of the data increases. This is expected for the iterative algorithms but not for the one-shot algorithms because they improved their performance when the underlying factors because more sparse in Figures 8.3(a) and 8.3(b). We hypothesize that this is because the one-shot algorithms are designed to recover the ground-truth factors in the near separable context, and the sparsity of those factors may make them more recoverable. Conversely, in the non-near-separable case there are no ground-truth matrices to recover, so the algorithms may take on unintended behavior that draws them further from the NMF solution when X is sparse.

8.4 Heavy-Tailed Data

In this section we test NMF algorithm performance on data sampled from one of the two heavy-tailed distributions that have yielded the most interesting results thus far

- the Pareto distribution. In Sections 3.5 and 6.1.6, we observed that data sampled from the Pareto distribution resulted in NMF error curves that choppily descended towards relatively low error rates, while ANLS BPP significantly outperformed HALS and MU.

Like we did for sparse data, we generate two types of heavy-tailed datasets, one which is near-separable and one which is not. We form the latter by sampling independently from the Pareto distribution, and putting the results in the matrix X. We plot the NMF algorithm error curves yielded by tests on matrices X generated in this manner in Figure 8.4. Meanwhile, we form the near-separable dataset $\tilde{X}$ by first sampling each of the ground-truth factors $W^\natural \in \mathcal{R}^{p\times r}$ and $H^\natural \in \mathcal{R}^{r\times n}$ from the Pareto distribution, then overwriting r randomly chosen columns in $H^\natural$ with standard basis vectors $H^\natural$ such that now $H^\natural = [I_r, H^{\natural'}]\Pi$ for some permutation Π. We then form the noise matrix N, scale $X = W^\natural H^\natural$ and N, and add these two matrices together, as we have done in the previous two sections. Again, because of the noise and the factor multiplication that formed it, $\tilde{X}$ is not necessarily heavy-tailed; what we are really testing here is the ability of the NMF algorithms to recover heavy-tailed factors. We plot the results of the tests on $\tilde{X}$ in Figure 8.4.

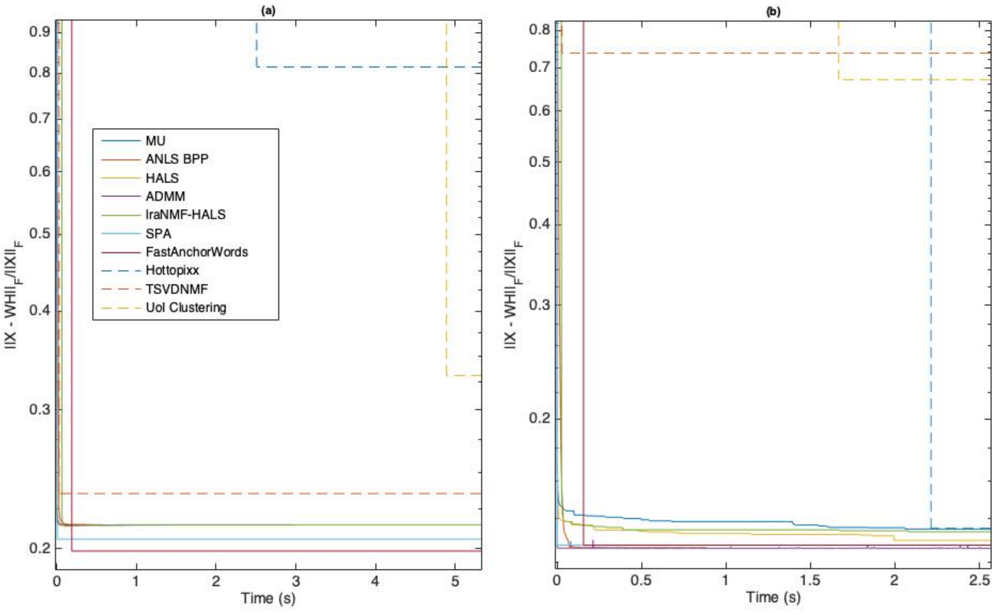


Figure 8.4: NMF relative error vs time for 10 different algorithms on data generated from Pareto distribution that is (a) near-separable and (b) non-near-separable.

To a large extent the results here tell a similar story as the results on sparse data in the previous section. When the dataset is near-separable and the ground-truth factors have some structure to them, such as being highly sparse or having each element generated from the same heavy-tailed distribution, TSVDNMF, SPA, and FastAnchorWords are relatively effective. SPA and FastAnchorWords perform especially well, reach lower relative errors than any other algorithm.

With that being said, it is worth noting that most algorithms perform better on the non-near-separable heavy-tailed data than on the near-separable data. The most stark example of this is Hottopixx dramatically increasing its accuracy, yielding a relative error of only 0.22 when $\tilde{X}$ is not near-separable, while it had a relative error near 0.8 when the data was near-separable. The one algorithm to buck this trend is TSVDNMF, which decreases in performance drastically once the data is not near-separable (in this case its relative error is 0.75). The fact that TSVDNMF is the only algorithm to exhibit this decline is surprising because the near-separable data has a ground truth solution in it (albeit obscured by small noise), whereas the non-near-separable data likely has a ground-truth solution covered in very significant noise, which suggests that the algorithms should be able to approximate the ground-truth solution in the former case but not reach as nearly an accurate factorization in the the latter. This is supported by the results in the previous section, in which the error drastically increases for all algorithms once the data is no longer near-separable. We are unsure of why the results exhibit this behavior on sparse data but not on heavy-tailed data.

Lastly, in a mostly unrelated note, we highlight that $UoI\text{-}NMF_{cluster}$ takes approximately three times as long to execute when the data is near-separable than when it is not. This is consistent with the results from the previous section - when the data was near-separable with sparse factors (and when it is 95% sparse and not near separable) the algorithm takes much longer to execute than when it is not near-separable and 50% sparse. Notably, $UoI\text{-}NMF_{cluster}$ did not take very long to execute when the data was dense and light-tailed in Section 8.2, so we are unsure why qualities of the underlying factors in the data affect the runtime so drastically. Perhaps it is due to ill-conditioning that slows down the execution of the many NNLS subproblems the algorithm must solve, but this is an area for further investigation.

8.5 Swimmer Dataset

In this section NMF algorithm performance is tested on the Swimmer dataset presented in [28]. The Swimmer dataset is composed of images of stick figures with a fixed torso and four limbs, each of which may be in four possible positions. Thus there are $4^4 = 256$ images in the dataset. We use a version of the dataset in which the zero pixels around each stick figure have been truncated such that each image is (20-by-11)[1], and display a few sample images in Figure 8.5. For reference, the Swimmer dataset is 88.12% sparse.

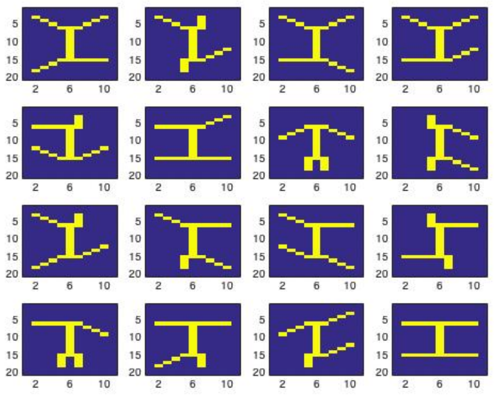


Figure 8.5: NMF relative error vs time on CBCL dataset (a) with and (b) without noise.

This dataset is not quite separable because no stick figure can be formed from additive combinations of the other stick figures. Nevertheless, it does have an efficient and interpretable parts-based representation. Consider that the only variant between the figures is the position of the limbs. This implies that a basis of 16 images, each composed of a limb in one of the 16 possible limb positions, would contain all of the Swimmer images in its convex hull.

Thus, when we run the NMF algorithms on the Swimmer dataset with factorization rank $r = 16$, we expect that the heuristics will find a close to exact solution (since one exists) whereas the recent algorithms that build W from examples in the dataset will not find an accurate solution, since no set of examples can additively reconstruct the others. We plot the results of our tests on both the noiseless and noisy Swimmer database in Figure 8.6.

Our predictions about algorithm performance in the noiseless case are partially realized in the data. The recent algorithms which form their estimates of W from columns of the dataset yield less accurate reconstructions than the other algorithms, with the exception of $UoI\text{-}NMF_{cluster}$, which performs similarly poorly. Meanwhile, ANLS BPP and ADMM yield very accurate solutions, with ANLS BPP eventually converging rapidly to a nearly perfectly accurate solution. Indeed, ANLS BPP recovers the bases we expected it would: from Figure 8.7, we see that each basis vector

[1]We obtained this dataset from Nicolas Gillis' website: `https://sites.google.com/site/nicolasgillis/code`.)

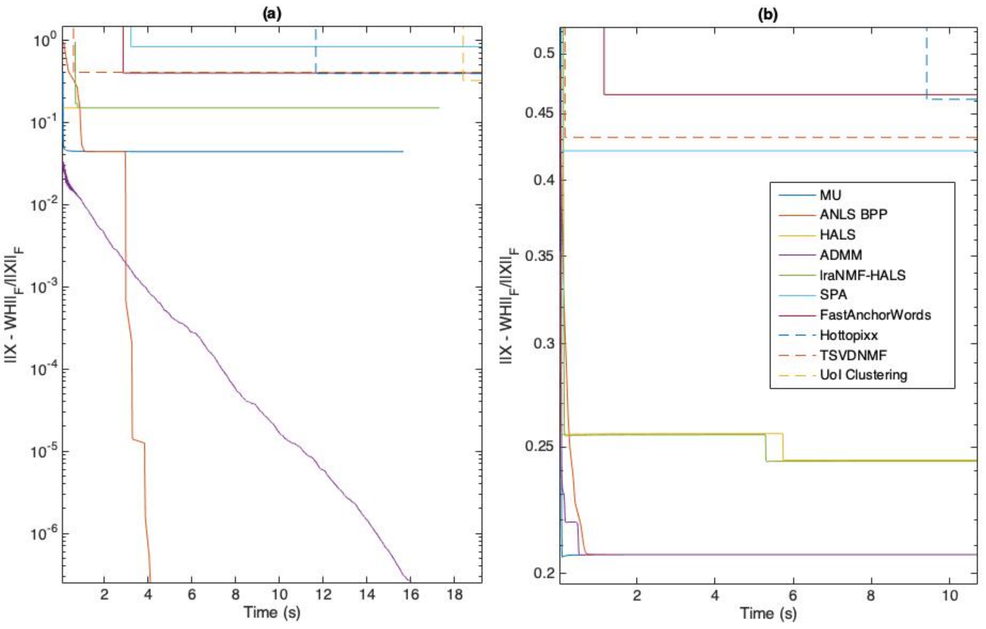


Figure 8.6: NMF relative error vs time on Swimmer dataset (a) without and (b) with noise.

recovered by ANLS BPP contains a different limb position. Four of the basis figures contain the fixed torso, and these four figures all have the same limb (but in different positions), which makes sense because every figure in the database must contain that limb, so to reconstruct any figure from the database one of those four basis figures must be involved, meaning there will be one instance of the torso in the additive combination.

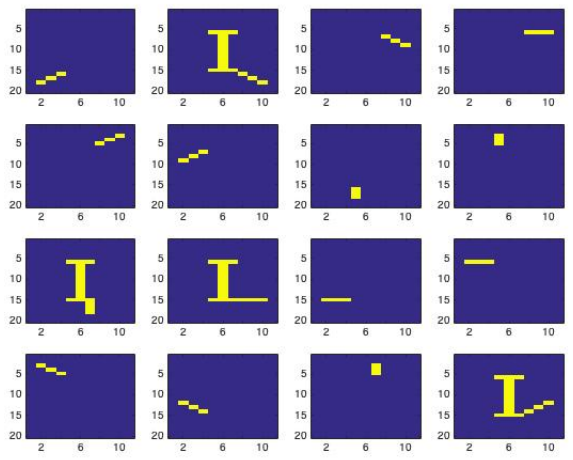


Figure 8.7: Bases returned by ANLS BPP for the Swimmer dataset.

However, the behavior of the other heuristics does not meet our expectations. They reach local minima that are more accurate than the one-shot algorithms, but still highly sub-optimal. We suspect that this behavior is because although perfectly accurate global optima exist, they are scarce in the set of all possible rank-r nonnegative factorizations, making HALS, lraNMF-HALS, and MU more likely to find a local minimum. For reasons that merit further investigation, ANLS BPP and ADMM are much better at finding a global minimum. Nevertheless, it makes sense that HALS and lraNMF-HALS perform similarly because the dataset is low rank.

In the noisy case, the added noise fits the same model as the noise model used in the previous sections: each element of the noise matrix N is first generated from $\sim |\mathrm{N}(0,1)|$, then N is normalized such that its Frobenius norm is equal to 1/4th of that of the data matrix. The results mostly match what we would expect. Each of the iterative heuristics performs worse, and ANLS BPP and ADMM no longer converge to perfect solutions, which makes sense because of the noise. Likewise, most of the one-shot algorithm still perform poorly, but we are unsure why $UoI\text{-}NMF_{cluster}$ takes so long to execute (23 seconds on average) and SPA improves its performance so drastically. The behavior of SPA suggests that added noise may be used to improve NMF algorithm performance, as we suggested in Section 7.9. Further investigation into this strategy is necessary.

8.6 Summary of Major Findings

Here we review the most significant results of this chapter:

- The NMF heuristics, both old and new, often outperform the new algorithms with provable performance guarantees, even on datasets for which those performance guarantees apply.

- ADMM is almost always the most effective method for solving NMF for the types of datasets we tested.

- The most recent algorithms are generally more robust to noise than the older heuristics. This is especially true for TSVDNMF, which is the most effective method in the high-noise, near-separable setting.

- TSVDNMF, SPA and FastAnchorWords perform better both when the ground-truth factors are more sparse and when they are drawn from a heavy-tailed distribution in the near-separable setting, especially FastAnchorWords. These algorithms, like the rest of the algorithms, perform worse when the data matrix is more sparse in the non-near-separable setting. However, most algorithms perform better for non-near-separable heavy-tailed data than near-separable heavy-tailed data, especially Hottopixx, which exhibits a drastic increase in accuracy.

- $UoI\text{-}NMF_{cluster}$ takes much longer to execute when the data is near-separable and heavy-tailed and very sparse than when it is not, while the other algorithms do not demonstrate the same variation in execution time.
- ANLS BPP and ADMM are the only algorithms to find an exact nonnegative factorization of the noiseless Swimmer dataset, and SPA finds a much more accurate solution when the dataset is noisy than when it is not.

Chapter 9

Conclusion

9.1 Summary of Results

In this work we provide an overview of and thoroughly test fourteen of the most prominent NMF algorithms and twelve of the most important initialization strategies on both synthetic and real data. The algorithms we investigate range from older, fundamental heuristics that are simple in structure and have no provable performance guarantees but tend to perform well in practice to more recent, complex algorithms that have provable performance guarantees under certain assumptions about the data and noise.

The central and encompassing conclusion from this work is that strategies to solve NMF vary significantly in effectiveness across the type of data being factored and the parameters of the problem. In Chapter 3, we showed that ALS, ANLS BPP, HALS and MU all perform the best out of the seven older heuristics we tested for factoring some type(s) of data. In Chapter 6, we demonstrated that the choice of initialization strategy can have a significant impact on NMF algorithm performance, especially for PGD and MU, and each of the twelve initialization strategies is among the best and/or worst choices for some type(s) of data. In Chapter 8, we showed that algorithm performance varies substantially depending on the data being factored, among not only recent heuristics but also the most recent NMF algorithms with provable guarantees. Moreover, we found that some NMF algorithms without performance guarantees often compute more accurate nonnegative factorizations than the algorithms with performance guarantees, again depending on the characteristics of the matrix being factored. These results emphasize that the algorithm and initialization used for NMF must be chosen wisely based on the problem setting, and our test results provide a guide for making that choice. Additionally, we showed that although multiple NMF algorithms failed to reveal topic information from a particular educational dataset, if the topics had been a non-trivial factor in whether students answered the questions correctly, the NMF algorithms would have recovered them from the data, highlighting the power of NMF as a data analysis tool in important applications.

9.2 Future Work

We have tried to expose the reader to as many algorithmic approaches and challenges to NMF as possible in this work, while recognizing that NMF is broad enough that it is impossible to address all of its research avenues in one paper. Thus, in our suggestions for future work we focus on NMF developments relevant to our work that we deem particularly important.

- First, further analysis and testing of the NMF algorithms, especially the recent algorithms, is necessary. For example, the promising performance of ADMM and the innovative intuition behind $UoI\text{-}NMF_{cluster}$ merit deeper investigation.
- Second, the development of more application-focused objective functions and algorithms is welcome. We saw in our application of NMF to educational data that a smaller relative error in the factorization does not necessarily yield a greater question clustering accuracy. Perhaps an algorithm that directly tried to yield an NMF with clustered weights would be more effective in this context than the NMF algorithms that try to minimize the NMF objective function. $UoI\text{-}NMF_{cluster}$ is an NMF algorithm that does not intentionally try to minimize the NMF objective function, but it does not provide measures of stability and interpretability which its motivation is to minimize [71]. If application-focused objectives are rigorously constructed, and algorithms are developed to minimize them, this could make the NMF solution more identifiable and effective.
- On a related note, identifiability deserves additional study. Since the 2003 Donoho and Stodden paper [28], the only significant extension of our knowledge of classes of identifiable matrices has been the result in [34], which showed that not only are separable matrices identifiable, but subset-separable (a broader notion of separable) matrices are as well. Existing theory tells us that an optimal solution (W, H) is unique only up to some orthogonal transformation(s), which are not necessarily permutations. Yet we observed during our testing that the Q-matrices generated by distinct NMF algorithms acting on the X_{NI} dataset

were approximately equal up to column shuffling. X_{NI} is likely not separable or even subset-separable, since the NMF algorithms did not yield solutions with errors going to zero, which suggests that perhaps there are other classes of matrices that yield identifiable nonnegative factorizations besides the known classes. Future research may aim at determining more of these classes.

- Another direction of theory-focused research is to further explore the optimization landscape of NMF. There has been much investigation into NMF from a geometric standpoint, but seemingly less from an optimization outlook. Meanwhile, results have recently been obtained describing the optimization landscape of similar low-rank matrix approximation problems; see especially the paper due to Ge et al. showing that all local minima of certain nonconvex low rank problems are global minima [33]. Although the NMF objective function is quadratic in W or H with the other matrix fixed, the argument in [33] do not apply to NMF because of its optimization over two matrices and the nonnegativity constraint on them. However, perhaps a related argument holds for NMF, and comes with a corresponding saddle-point escaping algorithm.

- Lastly, if there is one encapsulating takeaway from this paper, it is that NMF algorithms and initialization techniques vary significantly in effectiveness across different types of data. One natural response to this is to gather information about when some methods perform better than others so that practitioners may make the best decision based on this information, as we have done here. But what if we could automate and optimize the decision making process? We envision the development of a model that can predict the optimal NMF algorithm to use for a given NMF problem instance, which includes the data to be factored and the application the factorization will be used for. Meta learning has been an active area of research recently [50], but to the best of our knowledge it has yet to be applied to NMF beyond trial-and-error based techniques (e.g. [78, 70]). One option is to train neural networks to learn the optimal mappings from problem instances to solvers. However, this would likely require an extremely large amount of data, so perhaps a better option is to quantize the decision making process with a decision tree model, and have each node in the tree correspond to a different characteristic of the dataset, such as the characteristics we tested in this paper: sparsity, condition number, distribution and more.

Appendix A

MATLAB Code

This appendix contains our implementations of all the algorithms and initialization techniques discussed in this paper, as well as a representative sample of the code we used for testing. All of our implementations are adapted from the papers referenced in this thesis.

A.1 Older Heuristics and Educational Data Case Study (Chapters 2-4)

```
function [W, H, e, t, time] = twoBlockCD_ALS(X,r,T,init,W_init,H_init,epsilon,X_)

% Two Block Coordinate Descent - Alternating Least Squares
% Author: Liam Collins

% Input: X : ground-truth or noisy ground-truth nonnegative matrix in R^{pxn}
%        r : factorization rank
%        T : maximum number of iterations to complete
%        init : dictates initialization method
%        W_init, H_init : initial estimates for W, H
%        epsilon : error thresholding constant for optional stopping condition
%        X_ : ground-truth nonnegative matrix in R^{pxn}
%
% Output: (W, H) >= 0: A rank-r NMF of X \approx WH
%          W \in R^(pxr), H \in R^(rxn)
%          e : Tx1 Vector of relative errors ||X - WH||_F/||X||_F at each iteration
%          t : Number of iterations executed
%          time : Tx1 Vector of the cumulative time at which each iteration starts

normX = norm(X_,'fro');
[p,n] = size(X);
tic;

if init == 0
    W = W_init;
    H = H_init;
    %[W,H] = INIT_random(p,r,n);

% Subtractive clustering
elseif init == 1
    [W,H] = INIT_subclustering(X,r,n);

% NNDSVD
```

```
elseif init == 2
    [W,H] = INIT_NNDSVD(X,r);

% random Xcol
elseif init == 3
    [W,H] = INIT_Xcol(X,r,n);

% random-C
elseif init == 4
    [W,H] = INIT_randomC(X,r,n);

% co-occurrence
elseif init == 5
    [W,H] = INIT_cooccurrence(X,p,r,n);

% k-means with random column initialization
elseif init == 6
    [W,H] = INIT_kmeans(X,p,r,n,0,0,50);

% spherical k-means with random column initialization
elseif init == 7
    [W,H] = INIT_sphericalKmeans(X,p,r,n,0,0,50);

% PCA initilization
elseif init == 8
    [W,H] = INIT_PCA(X,r,n);

% SVD-NMF
elseif init == 9
    [W,H] = INIT_SVD_NMF(X,r);

% fuzzy c-means
elseif init == 10
    [W,H] = INIT_fuzzyMeans(X,p,r,n,0,0,50);

% NNDSVD with low-rank correction
elseif init == 11
    [W,H] = INIT_NNDSVD_LRC(X,r);

% given init
else
    W = W_init;
    H = H_init;
end

e = zeros(T,1);
t = 1;

% start timer
lr_time = 0;
time = zeros(1,T);
first = 1;
while (t <= T)
    % do not include time to calculate error in timing measurements
    % properly scale W
    alpha = (sum(sum(X*H'.*W)))/(sum(sum((W'*W).*(H*H'))));
    W = alpha*W;

    %dt = toc(stall);
    pt = toc;
    e(t) = norm(X_ - W*H,'fro')/normX;
    %if (first == 1)
    %    time(t) = 0;
    %    first = 0;
    %else
        lr_time = lr_time + pt;
        time(t) = lr_time;
    %end
    tic;

    %gradW = (W*(H*H') - X*H')';
    %gradH = (W'*W)*H - W'*X;

    %if (t == 1)
    %    thresh = epsilon*norm([gradW, gradH],'fro');
    %end

    %pgrad = norm([gradW(W>0); gradH(H>0)],'fro');
    %if (pgrad < thresh)
    %    e(t+1:end) = e(t);
    %    break;
    %end

    M = (X*H')/(H*H' + 1e-6*eye(r));
    W = max(M,0);
    M = (X'*W)/(W'*W + 1e-6*eye(r));
    H = max(M,0)';
    t = t + 1;
end
```

```
function [W_, H_, e, tinit, time] = twoBlockCD_MU(X,r,T,init,W_init,H_init,epsilon,X_)

% Two Block Coordinate Descent with Multiplicative Updates
% Author: Liam Collins

% Input: X : ground-truth or noisy ground-truth nonnegative matrix in R^{pxn}
%        r : factorization rank
%        T : maximum number of iterations to complete
%        init : dictates initialization method
%        W_init, H_init : initial estimates for W, H
%        epsilon : error thresholding constant for optional stopping condition
%        X_ : ground-truth nonnegative matrix in R^{pxn}
%
% Output: (W, H) >= 0: A rank-r NMF of X \approx WH
%          W \in R^(pxr), H \in R^(rxn)
%          e : Tx1 Vector of relative errors ||X - WH||_F/||X||_F at each iteration
%          t : Number of iterations executed
%          time : Tx1 Vector of the cumulative time at which each iteration starts

normX = norm(X_,'fro');
[p,n] = size(X);
tic;
% for analysis
W_=0;
H_=0;

if init == 0
    [W,H] = INIT_random(p,r,n);

    % Subtractive clustering
elseif init == 1
    [W,H] = INIT_subclustering(X,r,n);

    % NNDSVD
elseif init == 2
    [W,H] = INIT_NNDSVD(X,r);
    W_ = W;
    H_ = H;

    % random Xcol
elseif init == 3
    [W,H] = INIT_Xcol(X,r,n);

    % random-C
elseif init == 4
    [W,H] = INIT_randomC(X,r,n);

    % co-occurrence
elseif init == 5
    [W,H] = INIT_cooccurrence(X,p,r,n);

% k-means with random column initialization
elseif init == 6
    [W,H] = INIT_kmeans(X,p,r,n,0,0,50);

% spherical k-means with random column initialization
elseif init == 7
    [W,H] = INIT_sphericalKmeans(X,p,r,n,0,0,50);

    % PCA initilization
elseif init == 8
    [W,H] = INIT_PCA(X,r,n);

% SVD-NMF
elseif init == 9
    [W,H] = INIT_SVD_NMF(X,r);

% fuzzy c-means
elseif init == 10
    [W,H] = INIT_fuzzyMeans(X,p,r,n,0,0,50);

% NNDSVD with low-rank correction
elseif init == 11
    [W,H] = INIT_NNDSVD_LRC(X,r);

    % given init
else
    W = W_init;
    H = H_init;

end

% properly scale W
alpha = (sum(sum(X*H'.*W)))/(sum(sum((W'*W).*(H*H'))));
W = alpha*W;

e = zeros(T,1);
t = 1;
thresh = 0;

% start timer
lr_time = 0;
time = zeros(1,T);
first = 1;
tinit = toc;
tic;
while (t <= T)
    % do not include time to calculate error in timing measurements
    pt = toc;
    e(t) = norm(X_ - W*H,'fro')/normX;
    %if (first == 1)
    %    time(t) = 0;
    %    first = 0;
    %else
        lr_time = lr_time + pt;
        time(t) = lr_time;
    %end

    %gradW = (W*(H*H') - X*H');
    %gradH = (W'*W)*H - W'*X;

    %if (t == 1)
    %    thresh = norm([gradW; gradH'],'fro');
        %fprintf('init grad norm %f\n', thresh);
    %    thresh = epsilon*thresh;
    %end

    %pgrad = norm([gradW(W>0); gradH(H>0)],'fro');
    %if (pgrad < thresh)
    %    break;
    %end

    % stopping condition
    %projnorm = norm([gradW(W>0); gradH(H>0)],'fro');
    %norms(t) = projnorm;
    %if (projnorm < thresh)
        %fprintf('init grad norm %f\n', projnorm);
    %    e(t+1:end) = e(t);
    %    break;
    %end

    tic;

    % lower bound elements by delta to prevent them from going to (and
    % staying at) 0
    W = max(((W.*(X*H'))./(W*(H*H') + 1e-9*ones(p,r))),1e-12);
    H = max(((H.*(W'*X))./((W'*W)*H + 1e-9*ones(r,n))),1e-12);

    t = t + 1;
end
e(t:end) = e(t-1);
W=W;
```

```
function [W, H, e, tinit, time] = twoBlockCD_HALS(X,r,T,init,W_init,H_init,epsilon,X_);

% Two Block Coordinate Descent - Hierarchical Alternating  Least Squares
% Author: Liam Collins

% Input: X : ground-truth or noisy ground-truth nonnegative matrix in R^{pxn}
%        r : factorization rank
%        T : maximum number of iterations to complete
%        init : dictates initialization method
%        W_init, H_init : initial estimates for W, H
%        epsilon : error thresholding constant for optional stopping condition
%        X_ : ground-truth nonnegative matrix in R^{pxn}
%
% Output: (W, H) >= 0: A rank-r NMF of X \approx WH
%          W \in R^(pxr), H \in R^(rxn)
%          e : Tx1 Vector of relative errors ||X - WH||_F/||X||_F at each iteration
%          t : Number of iterations executed
%          time : Tx1 Vector of the cumulative time at which each iteration starts

normX = norm(X_,'fro');
[p,n] = size(X);
tic;

if init == 0
    [W,H] = INIT_random(p,r,n);

% Subtractive clustering
elseif init == 1
    [W,H] = INIT_subclustering(X,r,n);

% NNDSVD
elseif init == 2
    [W,H] = INIT_NNDSVD(X,r);

% random Xcol
elseif init == 3
    [W,H] = INIT_Xcol(X,r,n);

% random-C
elseif init == 4
    [W,H] = INIT_randomC(X,r,n);

% co-occurrence
elseif init == 5
    [W,H] = INIT_cooccurrence(X,p,r,n);

% k-means with random column initialization
elseif init == 6
    [W,H] = INIT_kmeans(X,p,r,n,0,0,50);

% spherical k-means with random column initialization
elseif init == 7
    [W,H] = INIT_sphericalKmeans(X,p,r,n,0,0,50);

% PCA initilization
elseif init == 8
    [W,H] = INIT_PCA(X,r,n);

% SVD-NMF
elseif init == 9
    [W,H] = INIT_SVD_NMF(X,r);

% fuzzy c-means
```

```
elseif init == 10
    [W,H] = INIT_fuzzyMeans(X,p,r,n,0,0,50);

% NNDSVD with low-rank correction
elseif init == 11
    [W,H] = INIT_NNDSVD_LRC(X,r);

% given init
else
    W = W_init;
    H = H_init;

end

% properly scale W
alpha = (sum(sum(X*H'.*W)))/(sum(sum((W'*W).*(H*H'))));
W = alpha*W;

e = zeros(T,1);
t = 1;
thresh = 0;
norms = zeros(1,T);

% start timer
lr_time = 0;
time = zeros(1,T);
first = 1;
tinit = toc;
tic;
while (t <= T)
    % do not include time to calculate error in timing measurements
    pt = toc;
    e(t) = norm(X_ - W*H,'fro')/normX;
        %if (first == 1)
    %    time(t) = 0;
    %    first = 0;
    %else
        lr_time = lr_time + pt;
        time(t) = lr_time;
    %end

    %gradW = (W*(H*H') - X*H');
    %gradH = (W'*W)*H - W'*X;

    %if (t == 1)
    %    thresh = norm([gradW; gradH'],'fro');
        %fprintf('init grad norm %f\n', thresh);
    %    thresh = epsilon*thresh;
    %end

    % stopping condition
    %projnorm = norm([gradW(W>0); gradH(H>0)],'fro');
    %norms(t) = projnorm;
    %if (projnorm < thresh)
    %    %fprintf('init grad norm %f\n', projnorm);
    %    e(t+1:end) = e(t);
    %    break;
    %end
    tic;

    A = X*H';
    B = H*H';
    for l = 1:r
        s = zeros(p,1);
        for k = 1:r
            if k ~= l
                s = s + W(:,k)*(B(k,l));
            end
        end
        Z = (A(:,l) - s)/B(l,l);
        W(:,l) = max(Z, 0);
        if W(:,l) == 0
            W(:,l) = 1e-16*max(W(:));
        end
    end

    A = X'*W;
    B = W'*W;
    for l = 1:r
        s = zeros(n,1);
        for k = 1:r
            if k ~= l
                s = s + H(k,:)'*(B(k,l));
            end
        end
        Z = (A(:,l) - s)/B(l,l);
        H(l,:) = max(Z, 0)';
        if isequal(H(l,:), zeros(size(H(l,:))))
            H(l,:) = 1e-16 + (H(l,:));
        end
    end
    t = t + 1;
end
```

```
function [W, H, e, tinit, time, numSubIterW, numSubIterH] = twoBlockCD_ANLS(X,r,T,T_ls,init,W_init
    ,H_init,epsilon,nnls,X_)

% Two Block Coordinate Descent - Alternating Nonnegative Least Squares
% Author: Liam Collins

% Input: X : ground-truth or noisy ground-truth nonnegative matrix in R^{pxn}
%        r : factorization rank
%        T : maximum number of iterations
%        T_ls : maximum number of subiterations allowed to solve NNLS subproblem
%        init : dictates initialization method
%        W_init, H_init : initial estimates for W, H
%        epsilon : error thresholding constant for optional stopping condition
%        nnls : dictates the method used to solve the NNLS subproblems
%               0 : Projected gradient descent (PGD)
%               1,2 : Active set (AS)
%               3 : Block principal pivoting (BPP)
%        X_ : ground-truth nonnegative matrix in R^{pxn}
%
% Output: (W, H) >= 0: A rank-r NMF of X \approx WH
%          W \in R^(pxr), H \in R^(rxn)
%          e : Tx1 Vector of relative errors ||X - WH||_F/||X||_F at each iteration
%          t : Number of iterations executed
%          time : Tx1 Vector of the cumulative time at which each iteration starts
%          numSubIterW, numSubIterH : number of NNLS subproblems in W, H which required >= 50
    subiterations

% This program calls
normX = norm(X_,'fro');
[p,n] = size(X);
tic;
%tic;

if init == 0
    [W,H] = INIT_random(p,r,n);

% Subtractive clustering
elseif init == 1
    [W,H] = INIT_subclustering(X,r,n);

% NNDSVD
elseif init == 2
    [W,H] = INIT_NNDSVD(X,r);

% random Xcol
elseif init == 3
    [W,H] = INIT_Xcol(X,r,n);

% random-C
elseif init == 4
    [W,H] = INIT_randomC(X,r,n);

% co-occurrence
elseif init == 5
```

```
    [W,H] = INIT_cooccurrence(X,p,r,n);

% k-means with random column initialization
elseif init == 6
    [W,H] = INIT_kmeans(X,p,r,n,0,0,50);

% spherical k-means with random column initialization
elseif init == 7
    [W,H] = INIT_sphericalKmeans(X,p,r,n,0,0,50);

% PCA initilization
elseif init == 8
    [W,H] = INIT_PCA(X,r,n);

% SVD-NMF
elseif init == 9
    [W,H] = INIT_SVD_NMF(X,r);

% fuzzy c-means
elseif init == 10
    [W,H] = INIT_fuzzyMeans(X,p,r,n,0,0,50);

% NNDSVD with low-rank correction
elseif init == 11
    [W,H] = INIT_NNDSVD_LRC(X,r);

% given init
else
    W = W_init;
    H = H_init;

end

W(find(W>1000)) = 0;
H(find(H>1000)) = 0;

% use MU for initialization
%[W, H, e, t, e_time, time] = twoBlockCD_MU(X,r,T,init,W_init,H_init,epsilon);

% properly scale W
alpha = (sum(sum(X*H'.*W)))/(sum(sum((W'*W).*(H*H'))));
W = alpha*W;

e = zeros(T,1);

t = 1;
thresh_w = 0;
thresh_h = 0;

gradW = W*(H*H') - X*H';
gradH = (W'*W)*H - W'*X;
thresh_w = 100*epsilon*norm([gradW;gradH'],'fro');
thresh_h = thresh_w;
numSubIterW = 0;
numSubIterH = 0;

% start timer
lr_time = 0;
time = zeros(1,T);
first = 1;
tinit = toc;
tic;
while (t <= T)
    % do not include time to calculate error in timing measurements
    %gradW = W*(H*H') - X*H';
    %gradH = (W'*W)*H - W'*X;

    %norm(X - W*H,'fro')/normX
    %dt = toc(stall);
    pt = toc;
    lr_time = lr_time + pt;
    time(t) = lr_time;
    e(t) = norm(X_ - W*H,'fro')/normX;
    if (e(t) < 10^-9)
```

```
        break;
    end
        %if (first == 1)
    %    time(t) = 0;
    %    first = 0;
    %else
        lr_time = lr_time + pt;
        time(t) = lr_time;
    %end

    % calculate threshold for stopping condition
    %if (first == 1)
     %   thresh_w = epsilon*norm([gradW;gradH'],'fro');
     %   thresh_h = thresh_w;
     %   thresh = epsilon*norm([gradW; gradH'],'fro');
     %   first = 2;
     %   lr_time = 0;
     %   tic;
     %   continue;
    %end

    % check stopping condition
    %pgrad = norm([gradW(W>0); gradH(H>0)],'fro');
    %if (pgrad < thresh)
    %    break;
    %end
    tic;

    if nnls == 0
        [W,gradW,t_w] = solveNNLS_PGD(X',W',H',T_ls,thresh_w);
        if (t_w >= 50)
            numSubIterW = numSubIterW + 1;
        end
    elseif nnls == 1
        [W,gradW,t_w] = solveNNLS_AS(X',H',r,p,T_ls);
        if (t_w >= 50)
            numSubIterW = numSubIterW + 1;
        end
    elseif nnls == 2
        [W,gradW,t_w] = solveNNLS_AS(X',H',r,p,T_ls);
        if (t_w >= 50)
            numSubIterW = numSubIterW + 1;
        end
    else
        [W,gradW,t_w] = solveNNLS_BP(X',H',r,p,T_ls);
        if (t_w >= 50)
            numSubIterW = numSubIterW + 1;
        end
    end
    W = W';
    gradW = gradW';

    if (t_w == 1)
        thresh_w = thresh_w/10;
    end

    if nnls == 0
        [H,gradH,t_h] = solveNNLS_PGD(X,H,W,T_ls,thresh_h);
        if (t_h >= 50)
            numSubIterH = numSubIterH + 1;
        end
    elseif nnls == 1
        [H,gradH,t_h] = solveNNLS_AS(X,W,r,n,T_ls);
        if (t_h >= 50)
            numSubIterH = numSubIterH + 1;
        end
    elseif nnls == 2
        [H,gradH,t_h] = solveNNLS_AS(X,W,r,n,T_ls);
        if (t_h >= 50)
            numSubIterH = numSubIterH + 1;
        end
    else
        [H,gradH,t_h] = solveNNLS_BP(X,W,r,n,T_ls);
        if (t_h >= 50)
```

```
            numSubIterH = numSubIterH + 1;
        end
    end

    if (t_h == 1)
        thresh_h = thresh_h/10;
    end

    t = t + 1;
end
e(t-1:end) = e(t-1);
time(t-1:end) = time(t-1);
```

```
function [H,grad,t] = solveNNLS_PGD(X,H,W,T,thresh)

% Solve nonnegative least squares subproblem with projected gradient
% descent
% Author: Liam Collins
%
% Input: X : ground-truth or noisy ground-truth nonnegative matrix in R^{pxn}
%        W, H : current estimates of W, H (or H^T, W^T) where H
%               is the matrix being optimized and W is constant
%        T : maximum number of iterations allowed
%        thresh : threshholding coefficient for stopping condition
%
% Output: H : matrix in R^{rxn} that solves the NNLS subproblem in H (or
%             W^T) exactly
%         grad : gradient of f w.r.t. H (or W^T)
%         t : number of iterations executed
%
% Implementation of the algorithm described by Lin in the paper:
% https://www.csie.ntu.edu.tw/~cjlin/papers/pgradnmf.pdf

WtW = W'*W;
WtX = W'*X;

[nn, mm] = size(WtX);
T = 50*nn;

beta = 0.1;
sigma = 0.01;
t = 1;
alpha = 1;
while (t <= T)

    grad = WtW*H - WtX;
    pgradH = norm(grad(grad < 0 | H >0), 'fro');
    if (pgradH <= thresh)
        break;
    end

    H_old = H;
    H_new = max(H - alpha*grad, 0);

    r = 0;

    if (1-sigma)*sum(sum(grad.*(H_new-H))) + 0.5*sum(sum((H_new-H).*((WtW)*(H_new-H)))) <= 0
        while (1-sigma)*sum(sum(grad.*(H_new-H))) + 0.5*sum(sum((H_new-H).*((WtW)*(H_new-H)))) <=
             0 && ~isequal(H_old, H_new)
            r = r+1;
            alpha = alpha/beta;
            H_old = H_new;
            H_new = max(H - alpha*grad, 0);
            if r == 20
                break;
            end
        end
        alpha = alpha*beta;
        H = max(H - alpha*grad, 0);
    else
        while (1-sigma)*sum(sum(grad.*(H_new-H))) + 0.5*sum(sum((H_new-H).*((WtW)*(H_new-H)))) > 0
            r = r+1;
            alpha = alpha*beta;
            H_new = max(H - alpha*grad, 0);
            norm(grad,'fro');
            if r == 20
                break;
            end
        end
        H = H_new;
    end
    t = t+1;
end

end
```

```
function [H,grad,t] = solveNNLS_AS(X,W,r,n,T)

% Solves nonnegative least squares subproblem with active-set method
% Author: Liam Collins
%
% Implemntation of the algorithm described by Kim and Park in the paper:
% https://epubs.siam.org/doi/pdf/10.1137/07069239X
%
% Input: X : ground-truth or noisy ground-truth nonnegative matrix in R^{pxn}
%        W : current estimate of W (or H^T) where W is constant for NNLS subproblem
%        r : factorization rank (number of columns of W, rows of H)
%        n : number of columns in H
%        T : maximum number of iterations allowed
%
% Output: H : matrix in R^{rxn} that solves the NNLS subproblem in H (or
%             W^T) exactly
%         grad : gradient of f w.r.t. H (or W^T)
%         t : number of iterations executed

WtW = W'*W;
WtX = W'*X;

H = zeros(r,n);
t = 1;

% passive set
S = false(r,n);
% active columns
Ecols = find(any(~S));
w = WtX;

% going to need to ensure that w is accounted for
while (~isempty(Ecols))
    % T is the maximum number of iterations
    if t > 5*r
        break;
    end

    % step 3
    Z = zeros(r,length(Ecols));
    nfs = find(~S(:,Ecols));
    Z = solveNormalEqComb(WtW, WtX(:,Ecols), S(:,Ecols));

    Z(nfs) = 0;

    % step 4
    %need to do this for each column
    % indices in subS that are passive
    % indices in subS that are passive AND z neg -> infeas
    % Z < 0 --> implies index is in passive set
    badCols = find(any(Z < 0));
    goodCols = find(all(Z >= 0));
    if (~isempty(badCols))
        Zbad = Z(:,badCols);
        badColsInds = Ecols(badCols);
        G = H(:,badColsInds);
        Alphas = Inf*ones(r,length(badCols));
        badInds = find(Zbad < 0);
        Alphas(badInds) = G(badInds)./(G(badInds)-Zbad(badInds));
        [M,I] = min(Alphas);
        Alpha = repmat(M,r,1);
        H(:,badColsInds) = H(:,badColsInds) + Alpha.*(Zbad - H(:,badColsInds));
        for j = 1:length(M)
            H(I(j),badColsInds(j)) = 0;
            S(I(j),badColsInds(j)) = false;
        end
    end
    if (~isempty(goodCols))
        goodColsInds = Ecols(goodCols);
        H(:,goodColsInds) = Z(:,goodCols);
        w(:,goodColsInds) = WtX(:,goodColsInds)- WtW*H(:,goodColsInds);

        newBadInds = (w(:,goodColsInds) > 0) & ~S(:,goodColsInds);
        newBadCols = goodColsInds(any(newBadInds));
        % steps 1 and 2 in published algorithm
        if ~isempty(newBadCols)
            [M,I] = max(w(:,newBadCols).*~S(:,newBadCols));
            for i = 1:length(I)
                S(I(i),newBadCols(i)) = true;
            end
        end
        Ecols = [Ecols(badCols), newBadCols];
    end
    t = t+1;
end

grad = 0;

end
```

```
function [H,grad,t] = solveNNLS_BP(X,W,r,n,T)

% Solve nonnegative least squares subproblem with block principal pivoting
% Author: Liam Collins
%
% Implemntation of the algorithm described by Kim and Park in the paper:
% https://pdfs.semanticscholar.org/92b1/8501fb7989ffa277805c42e0cd396dd2423b.pdf
%
% Input: X : ground-truth or noisy ground-truth nonnegative matrix in R^{pxn}
%        W : current estimate of W (or H^T) where W is constant for NNLS subproblem
%        r : factorization rank (number of columns of W, rows of H)
%        n : number of columns in H
%        T : maximum number of iterations allowed
%
% Output: H : matrix in R^{rxn} that solves the NNLS subproblem in H (or
%             W^T) exactly
%         grad : gradient of f w.r.t. H (or W^T)
%         t : number of iterations executed

WtW = W'*W;
WtX = W'*X;

F = zeros(r,n);
H = zeros(r,n);
Y = -WtX;

alpha = 3*ones(n,1);
beta = (r+1)*ones(n,1);

I = find(sum(((H < 0) & F) | ((Y < 0) & ~F)));
grad = 0;

t = 1;
T = 50*r;

while (~isempty(I))
    % T is the maximum number of iterations
    if t > 5*r
        break;
    end
    %grad = WtW*H - WtX;

    I = find(sum(((H < 0) & F) | ((Y < 0) & ~F)));

    s = length(I);
    V = zeros(r,s);
    V_ = zeros(r,s);
    for k = 1:s
        j = I(k);
        summ = 0;
        for i = 1:r
            if (H(i,j) < 0 & F(i,j)) || (Y(i,j) < 0 & ~F(i,j))
                V(i,k) = 1;
                summ = summ + 1;
            end
        end

        % V_ includes info on the infeasible elements, not just columns
        if summ < beta(j)
            beta(j) = summ;
            alpha(j) = 3;
            V_(:,k) = V(:,k);
        elseif summ >= beta(j) && alpha(j) >= 1
            alpha(j) = alpha(j) - 1;
            V_(:,k) = V(:,k);
        else
            mx = r;
            for i_ = 1:r
                i = r-i_+1;
                if V(i,k) == 1
                    mx = i;
                    break;
                end
            end
            V_(mx,k) = 1;
        end
    end

    % Update F and G
    F_old = F;
    VV_ = zeros(r,n);
    VV_(:,I) = V_;
    F = (max(F_old-VV_,0) | (~F & VV_));

    H_f = solveNormalEqComb(WtW, WtX(:,I), F(:,I));
    H(:,I) = H_f;
    H(abs(H)<1e-12) = 0;
    Y(:,I) = WtW*H_f - WtX(:,I);
    Y(abs(Y)<1e-12) = 0;

    t = t+1;
end

end
```

```
function A = rand_cond(p,n,kappa)

% Returns a nonnegative matrix A in R^{pxn} with conditon number kappa

kappa = round(kappa);

A = abs(randn(p,n));
[U,S,V] = svd(A,'econ');
s = diag(S);
cond(A)
if cond(A) > kappa
    %s = s(end)*(1 + ((kappa-1))*(s-s(end))/(s(1)-s(end)));
    error('Old condition number greater than new, matrix will not be nonnegative');
else
    s = s(1)*(1 - ((kappa-1)/kappa)*(s(1)-s)/(s(1)-s(end)));
end
S = diag(s);
A = U*S*V';
cond(A)
```

```
function [] = eduSim(Xni)

% Function to test question clustering accuracy for synthetic data
% generated by accounting for the influence of question difficulty
% student ability, skewness of the data towards right or wrong
% answers, and lecture topic.
% Author: Liam Collins

mu = -0.25;     % skewness of the data
sdTop = 1;      % influence of lecture topic
sdStud = 1;     % influence of student ability
sdItem = 1;     % influence question difficulty
sdNoise = 0;    % influence of noise

algs = 1;

p = 69;
n = 221;
r = 7;

%p = 40;
%n = 100;
%r = 4;

bins = [11,22,30,39,51,62,70];
%bins = [11,21,31,41];
%bins = [49,70];

bs = 0;
avg = 0;
acc = 0;

acc_ALS = 0;
acc_MU = 0;
acc_ANLS = 0;
acc_HALS = 0;

M = 40;
dev_ALS = zeros(M,1);
dev_MU = zeros(M,1);
dev_ANLS = zeros(M,1);
dev_HALS = zeros(M,1);

for mm = 1:M

P = zeros(p,n);
D = zeros(p,n);

beta_js = (sdItem)*randn(p,1) + mu;
beta_tops = (sdTop)*randn(r,1) + mu; % could add this for more obstruction

for s = 1:n
    beta_s = (sdStud)*randn + mu;
    k = 1;
    for i = 1:length(bins)
        beta_i = (sdTop)*randn + mu;
        for j = k:(bins(i)-1)
            beta_z = sqrt(sdNoise)*randn;
            bs = bs + (beta_s + beta_i + beta_js(j) + beta_z)/(p*n);
            P(j,s) = normcdf(beta_s + beta_i + beta_js(j) + beta_z, 0, 1);
            %P(j,s) = (12+(beta_s + beta_i + beta_js(j) + beta_z))/24;
        end
        k = bins(i);
    end
end

for i = 1:p
    for j = 1:n
        a = rand;
        if a < P(i,j)
            D(i,j) = 1;
        end
    end
end

%for tt = 1:n
%    D(:,tt) = D(:,tt)/norm(D(:,tt));
%end

```

```
figure(30)
clf
imagesc(D);

T = 1000;
Tbpp = 300;
init = 10;
epsilon = 0.0000001;
W_init = abs(randn(p,r));
H_init = abs(randn(r,n));

%D = Xni;
%inds = [6,8,15,16,19,21,25,26,27,29,44,46,49,50,55,57,58,59,60,68,69];
%swch = [51,52,53,54,56,61,62,63,64,65,66,67];
%for hh = 1:12
%    D([inds(hh) swch(hh)],:) = D([swch(hh) inds(hh)],:);
%end

for alg = 1:4

if alg == 1
    [W, H, e, t, e_time, time] = twoBlockCD_ALS(D,r,T,init,W_init,H_init,epsilon,D);

elseif alg == 2
    [W, H, e, t, e_time, time] = twoBlockCD_MU(D,r,2*T,init,W_init,H_init,epsilon,D);
    if mm == M
        figure(40);
        imagesc(W');
        xlabel('Question');
        ylabel('Lecture Topic');
        e(end)
    end

elseif alg == 3
    [W, H, e, t, e_time, time] = twoBlockCD_ANLS(D,r,Tbpp,50,init,W_init,H_init,epsilon,3,D);

else
    [W, H, e, t, e_time, time] = twoBlockCD_HALS(D,r,T,init,W_init,H_init,epsilon,D);
end

cont = zeros(r,r);
categ = 1;
k = 1;
for i = 1:length(bins)
    counts = zeros(1,r);
    for j = k:(bins(i)-1)
        [m,I] = max(W(j,:));
        counts(I) = counts(I) + 1;
    end
    cont(categ,:) = counts;
    categ = categ + 1;
    k = bins(i);
end

for i = 1:r
    if i == 1
        [m,I] = max(cont(:,1));
        cont([1 I],:) = cont([I 1],:);
    else
        benefit = zeros(r,1);
        for j = 1:(i-1)
            add = cont(j,i) - cont(j,j);
            [m,I] = max(cont(i:r,j));
            benefit(j) = add + m;
        end
        benefit(i:r) = (cont(i:r,i));
        [m,I] = max(benefit);
        if I > i
            cont([i I],:) = cont([I i],:);
        elseif I < i
            [m,I_] = max(cont(i:r,j));
            I_ = I_ + i - 1;
            if I_ == i
                cont([I I_],:) = cont([I_ I],:);
            else
                cont([I i],:) = cont([i I],:);
                cont([I I_],:) = cont([I_ I],:);
            end
        end
    end
end

acc_ = sum(diag(cont))/p;

if alg == 1
    acc_ALS = acc_ALS + acc_;
    dev_ALS(mm) = acc_;
elseif alg == 2
    acc_MU = acc_MU + acc_;
    dev_MU(mm) = acc_;
elseif alg == 3
    acc_ANLS = acc_ANLS + acc_;
    dev_ANLS(mm) = acc_;
else
    acc_HALS = acc_HALS + acc_;
    dev_HALS(mm) = acc_;
end

end

avg = avg+mean(mean(D));
acc = acc + acc_;
end

% for debugging
%bs/M;
%avg/M
%acc = acc/M;

acc_ALS = acc_ALS/M;
acc_MU = acc_MU/M
acc_ANLS = acc_ANLS/M;
acc_HALS = acc_HALS/M;

devs = zeros(4,1);
devs(1) = std(dev_ALS);
devs(2) = std(dev_MU)
devs(3) = std(dev_ANLS);
devs(4) = std(dev_HALS);

accs = [acc_ALS; acc_MU; acc_ANLS; acc_HALS];
figure(33)
clf
b1 = bar(1,accs(1),0.4);
hold on;
b2 = bar(2,accs(2),0.4);
b3 = bar(3,accs(3),0.4);
b4 = bar(4,accs(4),0.4);
names = {'ALS','MU','ANLS BPP','HALS'};
set(gca, 'XTick', [1 2 3 4])
set(gca, 'XTickLabel', names)

set(b1,'FaceColor',[224,47,15]/256);
set(b2,'FaceColor',[66,124,244]/256);
set(b3,'FaceColor',[160,24,62]/256);
set(b4,'FaceColor',[242,217,31]/256);
errorbar(1,accs(1),devs(1),'k');
errorbar(2,accs(2),devs(2),'k');
errorbar(3,accs(3),devs(3),'k');
errorbar(4,accs(4),devs(4),'k');
ylabel('Accuracy');
```

```
function [] = TEST_oldAlgs(Xni)

% Tests NMF algorithms over heavy-tailed data, noisy data, binary data,
% sparsity, dimension and debugging of algorithm execution on educational data
% Author: Liam Collins

```

```
p = 361;
r = 49;
n = 2429;
X = zeros(p,n);

maxes = zeros(70,7);
Winit = 0.1*ones(70,7);
bins = [11,22,30,39,51,62,71];
k = 1;
for i = 1:length(bins)
    for j = k:(bins(i)-1)
        Winit(j,i) = 1;
    end
    k = bins(i);
end

% ground truth solution
%aaa = abs(randn(p,r));
%bbb = abs(randn(r,n));
%X = aaa*bbb;

normX = norm(X,'fro');
epsilon = 0.0000001;

% all methods share same initialization
init = 11;
T = 120;  % max number of iterations
M = 1;  % number of trials
mm = 1; % number of variations of parameter to test
Tanls = 200;

ps = [20, 100, 250, 500];
ns = ps;
ks = [5,10,20,40];
zetas = [1/sqrt(2),1,sqrt(2),2];

%X = abs(randn(p,n));

figure(60)
clf;

for j = 1:1

p = 50;
n = 250;
%[p,n] = size(Xni);
%p = 250;
r = 10;
%n = ns(j);
%w = abs(randn(p,k));
%h = abs(randn(k,n));
nms = [0.25, 0.75, 0.9, 0.95];
zeta = zetas(j);

avgTime_MU = zeros(1,2.5*T);
avgTime_ALS = zeros(1,2*T);
avgTime_BP = zeros(1,Tanls);
avgTime_HALS = zeros(1,T);
avgTime_PGD = zeros(1,T);
avgTime_ALSPGD = zeros(1,Tanls);
avgTime_AS = zeros(1,Tanls);

avgE_MU = zeros(2.5*T,1);
avgE_ALS = zeros(2*T,1);
avgE_ANLS_BP = zeros(Tanls,1);
avgE_HALS = zeros(T,1);
avgE_PGD = zeros(T,1);
avgE_ALSPGD = zeros(Tanls,1);
avgE_ANLS_AS = zeros(Tanls,1);

for i = 1:M

    if (j < 0)
        X = random('Generalized Pareto',1,1,1,[p,n]);
    elseif (j < 0)
        X = random('Weibull',1,0.5,[p,n]);
    elseif (j < 0)
        X = exp(randn(p,n));
    elseif (j < 0)
        X = random('LogLogistic',1,1,[p,n]);
    end

    %X = zeros(p,n);
    %X = Xni;
    X = abs(randn(p,n));
    normX = norm(X, 'fro');

    %q = rand(p,n);
    %inds = find(q > nms(j));

    %X = abs(randn(p,n));
    %N = zeta*randn(p,n);
    %S = max(X+N,0);

    %inds = datasample(1:(p*n),nms(j),'Replace',false);
    %X(inds) = 1;

    W_init = abs(randn(p,r));
    inds = datasample(1:(p*r),200,'Replace',false);
    W_init(inds) = 0;

    H_init = abs(randn(r,n));
    inds = datasample(1:(n*r),1500,'Replace',false);
    H_init(inds) = 0;

    [W,H,e_MU,iter_MU, time_MU] = twoBlockCD_MU(X,r,2.5*T,init,W_init,H_init,epsilon,X);
    [W,H,e_ALS,iter_ALS, time_ALS] = twoBlockCD_ALS(X,r,2*T,init,W_init,H_init,epsilon,X);
    [W,H,e_HALS,iter_HALS, time_HALS] = twoBlockCD_HALS(X,r,T,init,W_init,H_init,epsilon,X);
    [W,H,e_ALSPGD,iter_ALSPGD, time_ALSPGD] = twoBlockCD_ANLS(X,r,Tanls,50,init,W_init,H_init,
        epsilon,0,X);
    [W,H,e_ANLS_AS,iter_ANLS_AS, time_AS] = twoBlockCD_ANLS(X,r,Tanls,50,init,W_init,H_init,
        epsilon,2, X);
    [W,H,e_ANLS_BP,iter_ANLS_BP, time_BP] = twoBlockCD_ANLS(X,r,Tanls,50,init,W_init,H_init,
        epsilon,3,X);
    [W,H,e_PGD,iter_PGD, time_PGD] = projectedGD(X,r,T,init,W_init,H_init,epsilon,X);

    avgTime_MU = avgTime_MU + time_MU/M;
    avgTime_ALS = avgTime_ALS + time_ALS/M;
    avgTime_PGD = avgTime_PGD + time_PGD/M;
    avgTime_ALSPGD = avgTime_ALSPGD + time_ALSPGD/M;
    avgTime_HALS = avgTime_HALS + time_HALS/M;
    avgTime_AS = avgTime_AS + time_AS/M;
    avgTime_BP = avgTime_BP + time_BP/M;


    avgE_MU = avgE_MU + e_MU/M;
    avgE_ALS = avgE_ALS + e_ALS/M;
    avgE_ANLS_BP = avgE_ANLS_BP + e_ANLS_BP/M;
    avgE_ANLS_AS = avgE_ANLS_AS + e_ANLS_AS/M;
    avgE_HALS = avgE_HALS + e_HALS/M;
    avgE_PGD = avgE_PGD + e_PGD/M;
    avgE_ALSPGD = avgE_ALSPGD + e_ALSPGD/M;
end

figure(60)
%subplot(2,2,j);
semilogy(avgTime_MU, avgE_MU);
hold on
semilogy(avgTime_ALS, avgE_ALS);
hold on
semilogy(avgTime_HALS, avgE_HALS);
hold on
semilogy(avgTime_PGD, avgE_PGD);
hold on
semilogy(avgTime_ALSPGD, avgE_ALSPGD);
hold on
semilogy(avgTime_AS, avgE_ANLS_AS);
hold on
```

```
semilogy(avgTime_BP, avgE_ANLS_BP);
hold on
legend('MU','ALS','HALS','PGD','ANLS PGD','ANLS AS','ANLS BPP');
%legend('MU','ALS','HALS','ANLS AS','ANLS BPP');
xlabel('Time (s)');
ylabel('||X - WH||_F/||X||_F');
if (j == 1)
    title('(a)');
elseif (j == 2)
    title('(b)');
elseif (j == 3)
    title('(c)');
elseif (j == 4)
    title('(d)');
end

end
```

A.2 Initialization Techniques (Chapters 5-6)

```
function [W,H] = INIT_random(p,r,n)
W = rand(p,r);
H = rand(r,n);
```

```
function [W,H] = INIT_Xcol(X,r,n)
H = abs(rand(r,n));
k = round(n/10);        % parameter dictating how many columns to average over
p = length(X(:,1));
W = zeros(p,r);
for i = 1:r
    s = randi(n,1,k);
    W(:,i) = sum(X(:,s),2)/k;
end
```

```
function [W,H] = INIT_randomC(X,r,n)

H = abs(rand(r,n));
k = r;        % parameter dictating how many columns to average over
norms = zeros(1,n);
for i = 1:n
    norms(i) = norm(X(:,i));
end
[b, I] = sort(norms);
p = length(X(:,1));
W = zeros(p,r);
for i = 1:r
    s = randi(4*r,1,k) + n - 4*r;
    W(:,i) = sum(X(:,I(s)),2)/k;
end
```

```
function [W, H] = INIT_cooccurrence(X,p,r,n)
H = rand(r,n);
k = r;        % parameter dictating how many columns to average over
G = X*X';
norms = zeros(1,p);
for i = 1:p
    norms(i) = norm(G(:,i));
end
[b, I] = sort(norms);
W = zeros(p,r);
for i = 1:r
    s = randi(r,1,k) + p - r;
    W(:,i) = sum(G(:,I(s)),2)/k;
end
```

```
function [W,H,time,e] = INIT_kmeans(X,p,r,n,init,C_,T)

% initialize r centroids
% ensure that initial centroids are distinct
if init == 0
```

```
    cols = randi(n,1,r);
    while (~isequal(size(cols), size(unique(cols))))
        cols = randi(n,1,r);
    end
    C = X(:,cols);
elseif init == 1
    C = C_;
end

time = zeros(1,T);
e = zeros(1,T);

t = 1;
d = zeros(n,r);
lr_time = 0;
tic;
while t <= T
    pt = toc;
    theta = 0;
    % compute distances
    for j = 1:r
        for i = 1:n
            d(i,j) = norm(X(:,i) - C(:,j));
        end
    end
    [tmp, py] = min(d,[],2);
    for j = 1:r
        inds = find(py==j);
        nj = length(inds);
        for i = 1:nj
            theta = theta + norm(X(:,inds(i)) - C(:,j))/norm(X(:,inds(i)));
        end
    end
    e(t) = theta;

    if (t == 1)
        time(t) = 0;
    else
        lr_time = lr_time + pt;
        time(t) = lr_time;
    end
    tic;
    % compute distances
    for j = 1:r
        for i = 1:n
            d(i,j) = norm(X(:,i) - C(:,j));
        end
    end

    % compute new clusters
    [tmp, py] = min(d,[],2);

    % compute new centroids
    for j = 1:r
        s = zeros(p,1);
        inds = find(py==j);
        nj = length(inds);
        for i = 1:nj
            s = s + (X(:,inds(i)) - C(:,j));
        end
        if (nj ~= 0)
            s = s/nj;
        end
        C(:,j) = C(:,j) + s;
    end
    t = t + 1;
end

W = C;
% initialize H as solution to least squares problem
%H = max(inv(W'*W+0.001*eye(r))*W'*X,0);
for j = 1:r
    for i = 1:n
        d(i,j) = X(:,i)'*C(:,j)/(norm(C(:,j))^2);%*norm(X(:,i)));
    end
end
H = d';
```

```
function [W,H,time,e] = INIT_sphericalKmeans(X,p,r,n,init,C_,T)

% Differences between k-means and spherical k-means: normalization and
% distance metric

% normalize each column
for j = 1:n
    nrm = norm(X(:,j));
    if (nrm ~= 0)
        X(:,j) = X(:,j)/nrm;
    end
end
normX = norm(X,'fro');

% initialize r centroids
% ensure that initial centroids are distinct
if init == 0
    cols = randi(n,1,r);
    while (~isequal(size(cols), size(unique(cols))))
        cols = randi(n,1,r);
    end
    C = X(:,cols);
elseif init == 1
    C = C_;
end


time = zeros(1,T);
e = zeros(1,T);

t = 1;
d = zeros(n,r);
lr_time = 0;
tic;
while t <= T
    pt = toc;
    theta = 0;
    % compute distances
    for j = 1:r
        for i = 1:n
            d(i,j) = X(:,i)'*C(:,j);
        end
    end
    [tmp, py] = max(d,[],2);
    for j = 1:r
        inds = find(py==j);
        nj = length(inds);
        for i = 1:nj
            theta = theta + X(:,inds(i))'*C(:,j);
        end
    end
    e(t) = theta;

    if (t == 1)
        time(t) = 0;
    else
        lr_time = lr_time + pt;
        time(t) = lr_time;
    end
    tic;

    % compute distances
    for j = 1:r
        for i = 1:n
            d(i,j) = X(:,i)'*C(:,j);
        end
    end

    % compute new clusters
    [tmp, py] = max(d,[],2);

    % compute new centroids
    for j = 1:r
        s = sum(X(:,find(py==j)),2);
        C(:,j) = s/norm(s);
    end
    t = t + 1;
end

W = C;
% initialize H as solution to least squares problem
%H = max(inv(W'*W + 0.01*eye(r))*W'*X,0);
for j = 1:r
    for i = 1:n
        d(i,j) = X(:,i)'*C(:,j);
    end
end
H = d';
```

```
function [W,H] = INIT_subclustering(X, r, n)

% Subtractive clustering
% this can also be used to suggest a value of r: Pr is first Pi s.t. Pi <
% 0.15*P1

%choose ra more methodically later
ra = 0.1;  % minimum acceptable distance between vectors of different clusters
rb = 1.5*ra;  %miminum acceptable distance between cluster prototypes

% normalize each column
for j = 1:n
    X(:,j) = X(:,j)/norm(X(:,j));
end
normX = norm(X,'fro');

p = length(X(:,1));
W = zeros(p,r);
H = zeros(r,n);

P = zeros(1,n);
for j = 1:n
    s = 0;
    for k = 1:n
        s = s + exp(-4*norm(X(:,j) - X(:,k))^2/ra^2);
    end
    P(j) = s;
end

for i = 1:r
    [P_star, I] = max(P);
    W(:,i) = X(:,I);
    s = 0;
    for j = 1:n
        P(j) = P(j) - P_star*exp(-4*norm(X(:,j)-X(:,I))^2/rb^2);
        if (i == 1)
            exp(-4*norm(X(:,j)-X(:,I))^2/rb^2);
        end
        % first it picks centroids close to the other ones, thats
        % why the exps are pretty large
        % then exps quickly get tiny, so it basically picks points
        % arbitrarily
        s = s+exp(-4*norm(X(:,j)-X(:,I))^2/rb^2);
    end
end

sigma2 = ra^2/8;
denoms = zeros(1,n);
for j = 1:n
    denom = 0;
    for i = 1:r
        denom = denom + exp(-0.5*norm(X(:,j)-W(:,i))^2/sigma2);
    end
    denoms(j) = denom;
end
```

```

for k = 1:r
    for j = 1:n
        num = exp(-0.5*norm(X(:,j)-W(:,k))^2/sigma2);
        H(k,j) = num/denoms(j);
    end
end
```

```
function [W,H,time,e] = INIT_fuzzyMeans(X,p,r,n,init,C_,T)

m = 2;

% initialize r centroids
% ensure that initial centroids are distinct
if init == 0
    cols = randi(n,1,r);
    while (~isequal(size(cols), size(unique(cols))))
        cols = randi(n,1,r);
    end
    C = X(:,cols);
elseif init == 1
    C = C_;
end

C = max(randn(p,r) + mean(X,2)*ones(1,r),0);

time = zeros(1,T);
e = zeros(1,T);

t = 1;
d = zeros(n,r);
U = zeros(n,r);
U(:,1) = ones(n,1);
lr_time = 0;
tic;
while t <= T
    pt = toc;
    theta = 0;
    for j = 1:r
        for i = 1:n
            theta = theta + U(i,j)^m*norm(X(:,i) - C(:,j))^2;
        end
    end
    e(t) = theta;

    if (t == 1)
        time(t) = 0;
    else
        lr_time = lr_time + pt;
        time(t) = lr_time;
    end
    tic;

    % compute distances
    for j = 1:r
        for i = 1:n
            d(i,j) = norm(X(:,i)-C(:,j));
        end
    end

    % update u
    for j = 1:r
        for i = 1:n
            s = 0;
            for l = 1:r
                s = s+(d(i,j)/d(i,l))^(2/(m-1));
            end
            U(i,j) = 1/s;
        end
    end

    % compute new centroids
    for j = 1:r
        sNum = zeros(p,1);
        sDenom = 0;
        for i = 1:n
            sNum = sNum + U(i,j)^(m)*X(:,i);
            sDenom = sDenom + U(i,j)^(m);
        end
        C(:,j) = sNum/sDenom +randn(p,1)/10^6;
        %C(:,j) = C(:,j)/norm(C(:,j));
    end

    t = t + 1;
end

W = C+ 1*randn(p,r)/10^5;
w = norm(W,'fro')^2/r;
x = norm(X,'fro')^2/n;
H = U';
```

```
function [W,H] = INIT_PCA(X,r,n)
u = sum(X,2)/n;     %pn
one = ones(1,n);
A = X - u*one;      %pn
C = A*A';           %p^2 n

[V,D] = eig(C);
d = diag(D);

%U = A*sqrt(inv(D))*V;

[d,I] = sort(d);  %plogp
d = flip(d);
I = flip(I);
B = V(:,I(1:r));
G = B'*(A);         %prn

B = abs(B);          %pr + rn
G = abs(G);

H = min(1, G);
W = min(1, B);
```

```
function [W,H] = INIT_NNDSVD(X, r)
    [U,S,V] = svds(X,r);
    p = length(X(:,1));
    n = length(X(1,:));
    W = zeros(p,r);
    H = zeros(r,n);
    W(:,1) = abs(sqrt(S(1,1))*U(:,1));
    H(1,:) = abs(sqrt(S(1,1))*V(:,1)');
    r_ = min(r,length(U(1,:)));
    for j = 2:r_
        x = U(:,j);
        y = V(:,j);
        xp = max(x,0);
        yp = max(y,0);
        xn = max(-x,0);
        yn = max(-y,0);
        xpnrm = norm(xp);
        ypnrm = norm(yp);
        mp = xpnrm*ypnrm;
        xnnrm = norm(xn);
        ynnrm = norm(yn);
        mn = xnnrm*ynnrm;
        if mp > mn
            u = xp/xpnrm;
            v = yp/ypnrm;
            sigma = mp;
        else
            u = xn/xnnrm;
            v = yn/ynnrm;
            sigma = mn;
        end
        W(:,j) = sqrt(S(j,j)*sigma)*u;
        H(j,:) = sqrt(S(j,j)*sigma)*v';
    end
    zX = find(X == 0);
    zW = find(W == 0);
    zH = find(H == 0);
    mu = mean(mean(X))/100;
    %while (length(zX)/(p*n) < length(zW)/(p*r) + length(zH)/(r*n))
    while (length(zX) < length(find(W*H==0)))

        if (length(zW) > 10)
            indsW = randperm(length(zW),10);
        else
            indsW = 1:length(zW);
        end
        if (length(zH) > 10)
            indsH = randperm(length(zH),10);
        else
            indsH = 1:length(zH);
        end
        for i = 1:length(indsW)
            W(zW(indsW(i))) = mu*rand;
        end
        for i = 1:length(indsH)
            H(zH(indsH(i))) = mu*rand;
        end
        zW = find(W==0);
        zH = find(H==0);
    end
    size(find(W==0))
    size(find(H==0))
```

```
function [W,H] = INIT_SVD_NMF(X, r)

% SVD then takes absolute value of factors, unlike NNDSVD
% Implementation by Liam Collins

[U,S,V] = svds(X,r);
r_ = min(length(U(1,:)),r);
p = length(X(:,1));
n = length(X(1,:));
W = zeros(p,r);
H = zeros(r,n);
W(:,1:r_) = abs(U);
H(1:r_,:) = abs(sqrt(S)*V');
```

```
function [W,H] = INIT_NNDSVD_LRC(X, r)

% NNSVD with correction to account for the rank of the approximation
% Due to Syed et al. in https://arxiv.org/pdf/1807.04020.pdf
% Implementation by Liam Collins

    k = ceil(r/2+1);
    [U,S,V] = svds(X,k);
    p = length(X(:,1));
    n = length(X(1,:));
    W = zeros(p,r);
    H = zeros(r,n);
    W(:,1) = abs(sqrt(S(1,1))*U(:,1));
    H(1,:) = abs(sqrt(S(1,1))*V(:,1)');
    r_ = min(r,2*length(U(1,:))-1);
    l = 2;
    for j = 2:r_
        x = U(:,l);
        y = V(:,l);
        xp = max(x,0);
        yp = max(y,0);
        xn = max(-x,0);
        yn = max(-y,0);

        if (mod(j,2) == 0)
            W(:,j) = sqrt(S(l,l))*xp;
            H(j,:) = sqrt(S(l,l))*yp';
        else
            W(:,j) = sqrt(S(l,l))*xn;
            H(j,:) = sqrt(S(l,l))*yn';
            l = l + 1;
        end
    end
    Xk = U*S*V';
    e_init = norm(Xk-W*H,'fro');
    e_new = e_init;
    t = 1;
    size(find(W==0))
    size(find(H==0))
    while(t < 50)
        e_old = e_new;
        [W,H] = twoBlockCD_HALS(Xk,r,1,12,W,H,0.000001,Xk);
        e_new = norm(Xk-W*H,'fro');
        if e_old - e_new < 0.01*e_init
            break;
        end
        t=t+1;
    end
    'lrc';
    size(find(W==0));
    size(find(H==0));
    size(find(Xk<0.000001));
    cond(Xk)
```

```
function [] = TEST_INIT_2()

% Tests NMF algorithms with varying initializations and Gaussian data.
% Author: Liam Collins

p = 361;
r = 49;
n = 2429;
X = zeros(p,n);

% construct X from CBLI face dataset
% used for debugging
for i = 1:0
    filename = strcat('./face/face', sprintf('%05d',i));
    filename = strcat(filename, '.pgm');
    A = imread(filename);
    X(:,i) = reshape(A,361,1);
    % normalize columns of X
    X(:,i) = X(:,i)/norm(X(:,i));
end

%X = rand_cond(50,250,10000);
%s = svds(X,100)
%sum(s(1:6))/sum(s)

%p = 11462;
%r = 49;
%n = 5810;
%X = csvread('./nips.csv', 2,2);
%X = X(1:p,:);
%normX = norm(X, 'fro');


p = 50;
r = 10;
n = 250;

epsilon = 0.0000001;

T = 100;  % max number of iterations
M = 20;  % number of trials

data = 1:(p*n);

figure(70)
clf

tinit_rand = 0;
tinit_xcol = 0;
tinit_randc = 0;
tinit_cooc = 0;
```

```
tinit_kmeans = 0;
tinit_skmeans = 0;
tinit_fmeans = 0;
tinit_subclus = 0;
tinit_pca = 0;
tinit_nndsvd = 0;
tinit_nndsvdlrc = 0;
tinit_svdnmf = 0;

for nmf = 1:4

if (nmf == 1 || nmf == 4)
    coef = 5;
elseif nmf == 3
    coef = 0.5;
else
    coef = 1;
end

avg_rand_time = zeros(1,coef*T);
avg_xcol_time = zeros(1,coef*T);
avg_randc_time = zeros(1,coef*T);
avg_cooc_time = zeros(1,coef*T);
avg_kmeans_time = zeros(1,coef*T);
avg_skmeans_time = zeros(1,coef*T);
avg_fmeans_time = zeros(1,coef*T);
avg_subclus_time = zeros(1,coef*T);
avg_pca_time = zeros(1,coef*T);
avg_nndsvd_time = zeros(1,coef*T);
avg_nndsvdlrc_time = zeros(1,coef*T);
avg_svdnmf_time = zeros(1,coef*T);

avg_rand_e = zeros(coef*T,1);
avg_xcol_e = zeros(coef*T,1);
avg_randc_e = zeros(coef*T,1);
avg_cooc_e = zeros(coef*T,1);
avg_kmeans_e = zeros(coef*T,1);
avg_skmeans_e = zeros(coef*T,1);
avg_fmeans_e = zeros(coef*T,1);
avg_subclus_e = zeros(coef*T,1);
avg_pca_e = zeros(coef*T,1);
avg_nndsvd_e = zeros(coef*T,1);
avg_nndsvdlrc_e = zeros(coef*T,1);
avg_svdnmf_e = zeros(coef*T,1);

for i = 1:M

    % RANK
    %U = abs(randn(p,10));
    %V = abs(randn(10,n));
    %X = U*V;
    %r = 15;

    % SPARSITY
    %X = abs(randn(p,n));
    %percSparse = 95;
    %numInds = percSparse*p*n/100;
    %inds = datasample(data,numInds,'Replace',false);
    %X(inds) = 0;

    % COND NUM
    %X = rand_cond(p,n,10000);

    % BINARY
    %q = rand(p,n);
    %inds = find(q > 0.5);
    %X = ones(p,n);
    %X(inds) = 0;

    % Heavy
    X = random('Generalized Pareto',1,1,1,[p,n]);
    %X = random('LogLogistic',1,1,[p,n]);

    %X = random('Weibull',1,0.5,[p,n]);

    %normX = norm(X, 'fro');

    %X = abs(randn(p,n));
    %normX = norm(X, 'fro');

    W_init = abs(randn(p,r));
    H_init = abs(randn(r,n));

    %'a'
    [W,H,rand_time,rand_e,tinit] = NMF(X,r,coef*T,50,12,W_init,H_init,epsilon,3,X,nmf);
    tinit_rand = tinit_rand+ tinit/(4*M);
%'b'
    [W,H,xcol_time,xcol_e,tinit] = NMF(X,r,coef*T,50,3,W_init,H_init,epsilon,3,X,nmf);
    tinit_xcol = tinit_xcol+ tinit/(4*M);
%'c'
    [W,H,randc_time,randc_e,tinit] = NMF(X,r,coef*T,50,4,W_init,H_init,epsilon,3,X,nmf);
    tinit_randc = tinit_randc+ tinit/(4*M);
%'d'
    [W,H,cooc_time,cooc_e,tinit] = NMF(X,r,coef*T,50,5,W_init,H_init,epsilon,3,X,nmf);
    tinit_cooc = tinit_cooc+ tinit/(4*M);
%'e'

    [W,H,kmeans_time,kmeans_e,tinit] = NMF(X,r,coef*T,50,6,W_init,H_init,epsilon,3,X,nmf);
    tinit_kmeans = tinit_kmeans+ tinit/(4*M);
%'f'
    [W,H,skmeans_time,skmeans_e,tinit] = NMF(X,r,coef*T,50,7,W_init,H_init,epsilon,3,X,nmf);
    tinit_skmeans = tinit_skmeans+ tinit/(4*M);
%'g'
    [W,H,fmeans_time,fmeans_e,tinit] = NMF(X,r,coef*T,50,10,W_init,H_init,epsilon,3,X,nmf);
    tinit_fmeans = tinit_fmeans+ tinit/(4*M);
%'h'
    [W,H,subclus_time,subclus_e,tinit] = NMF(X,r,coef*T,50,1,W_init,H_init,epsilon,3,X,nmf);
    tinit_subclus = tinit_subclus+ tinit/(4*M);
%'i'
    [W,H,pca_time,pca_e,tinit] = NMF(X,r,coef*T,50,8,W_init,H_init,epsilon,3,X,nmf);
    tinit_pca = tinit_pca+ tinit/(4*M);
%'j'
    [W_,H_,nndsvd_time,nndsvd_e,tinit] = NMF(X,r,coef*T,50,2,W_init,H_init,epsilon,3,X,nmf);
    tinit_nndsvd = tinit_nndsvd+ tinit/(4*M);
%'k'
    [W,H,nndsvdlrc_time,nndsvdlrc_e,tinit] = NMF(X,r,coef*T,50,11,W_init,H_init,epsilon,3,X,nmf);
    tinit_nndsvdlrc = tinit_nndsvdlrc+ tinit/(4*M);
%'l'
    [W,H,svdnmf_time,svdnmf_e,tinit] = NMF(X,r,coef*T,50,9,W_init,H_init,epsilon,3,X,nmf);
    %'m'
tinit_svdnmf = tinit_svdnmf+ tinit/(4*M);


    avg_rand_time = avg_rand_time + rand_time/M;
    avg_xcol_time = avg_xcol_time + xcol_time/M;
    avg_randc_time = avg_randc_time + randc_time/M;
    avg_cooc_time = avg_cooc_time + cooc_time/M;
    avg_kmeans_time = avg_kmeans_time + kmeans_time/M;
    avg_skmeans_time = avg_skmeans_time + skmeans_time/M;
    avg_fmeans_time = avg_fmeans_time + fmeans_time/M;
    avg_subclus_time = avg_subclus_time + subclus_time/M;
    avg_pca_time = avg_pca_time + pca_time/M;
    avg_nndsvd_time = avg_nndsvd_time + nndsvd_time/M;
    avg_nndsvdlrc_time = avg_nndsvdlrc_time + nndsvdlrc_time/M;
    avg_svdnmf_time = avg_svdnmf_time + svdnmf_time/M;

    avg_rand_e = avg_rand_e + rand_e/M;
    avg_xcol_e = avg_xcol_e + xcol_e/M;
    avg_randc_e = avg_randc_e + randc_e/M;
    avg_cooc_e = avg_cooc_e + cooc_e/M;
    avg_kmeans_e = avg_kmeans_e + kmeans_e/M;
    avg_skmeans_e = avg_skmeans_e + skmeans_e/M;
    avg_fmeans_e = avg_fmeans_e + fmeans_e/M;
    avg_subclus_e = avg_subclus_e + subclus_e/M;
    avg_pca_e = avg_pca_e + pca_e/M;
    avg_nndsvd_e = avg_nndsvd_e + nndsvd_e/M;
    avg_nndsvdlrc_e = avg_nndsvdlrc_e + nndsvdlrc_e/M;
    avg_svdnmf_e = avg_svdnmf_e + svdnmf_e/M;
```

```

end

figure(70)
subplot(2,2,nmf);
semilogy(avg_rand_time, avg_rand_e, '--');
hold on;
semilogy(avg_xcol_time, avg_xcol_e, '--');
hold on;
semilogy(avg_randc_time, avg_randc_e, '--');
hold on;
semilogy(avg_cooc_time, avg_cooc_e, '--');
hold on;
semilogy(avg_kmeans_time, avg_kmeans_e, '-.');
hold on;
semilogy(avg_skmeans_time, avg_skmeans_e, '-.');
hold on;
semilogy(avg_fmeans_time, avg_fmeans_e, '-.');
hold on;
semilogy(avg_subclus_time, avg_subclus_e, '-.');
hold on;
semilogy(avg_pca_time, avg_pca_e);
hold on;
semilogy(avg_nndsvd_time, avg_nndsvd_e);
hold on;
semilogy(avg_nndsvdlrc_time, avg_nndsvdlrc_e);
hold on;
semilogy(avg_svdnmf_time, avg_svdnmf_e);
hold on;
if nmf==1
    title('MU');
elseif nmf == 2
    title('HALS');
elseif nmf == 3
    title('ANLS BPP');
else
    title('PGD');
    legend('Random', 'Rand-Xcol','Rand-c','Co-occurrence','k-means', 'Spherical k-means','Fuzzy c-
        means','Sub. Clustering','PCA','NNDSVD','NNDSVD-LRC','SVD-NMF');
end
xlabel('Time (s)');
ylabel('Relative error');

end

tinit_rand
tinit_xcol
tinit_randc
tinit_cooc
tinit_kmeans
tinit_skmeans
tinit_fmeans
tinit_subclus
tinit_pca
tinit_nndsvd
tinit_nndsvdlrc
tinit_svdnmf

length(find(abs(W_)<0.0000001))
length(find(abs(H_)<0.0000001))
end
```

A.3 Recent Algorithms (Chapters 7-8)

```
function [Wp, Hp, e, time] = ADMM(X,r,T,init,W_init,H_init,epsilon,X_)

% Alternating Direction Method of Multipliers
% Author: Liam Collins

% Input: X : ground-truth or noisy ground-truth nonnegative matrix in R^{pxn}
%        r : factorization rank
%         T : maximum number of iterations to complete
%         init : dictates initialization method
%         W_init, H_init : initial estimates for W, H
%         epsilon : error thresholding constant for optional stopping condition
%         X_ : ground-truth nonnegative matrix in R^{pxn}
%
% Output: (W, H) >= 0: A rank-r NMF of X \approx WH
%           W \in R^(pxr), H \in R^(rxn)
%           e : Tx1 Vector of relative errors ||X - WH||_F/||X||_F at each iteration
%           t : Number of iterations executed
%           time : Tx1 Vector of the cumulative time at which each iteration starts

normX = norm(X_,'fro');
[p,n] = size(X);
tic;

if init == 0
    W = W_init;
    H = H_init;
    %[W,H] = INIT_random(p,r,n);

% Subtractive clustering
elseif init == 1
    [W,H] = INIT_subclustering(X,r,n);

% NNDSVD
elseif init == 2
    [W,H] = INIT_NNDSVD(X,r);

% random Xcol
elseif init == 3
    [W,H] = INIT_Xcol(X,r,n);

% random-C
elseif init == 4
    [W,H] = INIT_randomC(X,r,n);

% co-occurrence
elseif init == 5
    [W,H] = INIT_cooccurrence(X,p,r,n);

% k-means with random column initialization
elseif init == 6
    [W,H] = INIT_kmeans(X,p,r,n,0,0,50);

% spherical k-means with random column initialization
elseif init == 7
    [W,H] = INIT_sphericalKmeans(X,p,r,n,0,0,50);

% PCA initilization
elseif init == 8
    [W,H] = INIT_PCA(X,r,n);

% SVD-NMF
elseif init == 9
    [W,H] = INIT_SVD_NMF(X,r);

% fuzzy c-means
elseif init == 10
    [W,H] = INIT_fuzzyMeans(X,p,r,n,0,0,50);

% NNDSVD with low-rank correction
elseif init == 11
    [W,H] = INIT_NNDSVD_LRC(X,r);

% given init
else
    W = W_init;
    H = H_init;
end

e = zeros(T,1);
t = 1;

```

```
rho = 1;
lambdaW = zeros(p,r);
lambdaH = zeros(r,n);
Wp = zeros(p,r);
Hp = zeros(r,n);

% start timer
lr_time = 0;
time = zeros(1,T);
%first = 1;
while (t <= T)

    %dt = toc(stall);
    pt = toc;
    e(t) = norm(X_ - W*H,'fro')/normX;
    %if (first == 1)
    %    time(t) = 0;
    %    first = 0;
    %else
        lr_time = lr_time + pt;
        time(t) = lr_time;
    %end
    tic;

    %gradW = (W*(H*H') - X*H')';
    %gradH = (W'*W)*H - W'*X;

    %if (t == 1)
    %    thresh = epsilon*norm([gradW, gradH],'fro');
    %end

    %pgrad = norm([gradW(W>0); gradH(H>0)],'fro');
    %if (pgrad < thresh)
    %    e(t+1:end) = e(t);
    %    break;
    %end

    W = ((H*H' + eye(r))\(H*X' + Wp' - lambdaW'/rho))';
    H = (W'*W + eye(r))\(W'*X + Hp - lambdaH/rho);
    Wp = max(W+lambdaW/rho,0);
    Hp = max(H+lambdaH/rho,0);
    lambdaW = lambdaW + rho*(W-Wp);
    lambdaH = lambdaH + rho*(H-Hp);
    t = t + 1;
end
```

```
function [W, H, e, time] = lraNMF_HALS(X,r,T,init,W_init,H_init,epsilon,X_);

% Low rank approx NMF - Hierarchical Alternating Least Squares
% Author: Liam Collins

% Input: X : ground-truth or noisy ground-truth nonnegative matrix in R^{pxn}
%        r : factorization rank
%        T : maximum number of iterations to complete
%        init : dictates initialization method
%        W_init, H_init : initial estimates for W, H
%        epsilon : error thresholding constant for optional stopping condition
%        X_ : ground-truth nonnegative matrix in R^{pxn}
%
% Output: (W, H) >= 0: A rank-r NMF of X \approx WH
%          W \in R^(pxr), H \in R^(rxn)
%          e : Tx1 Vector of relative errors ||X - WH||_F/||X||_F at each iteration
%          t : Number of iterations executed
%          time : Tx1 Vector of the cumulative time at which each iteration starts

normX = norm(X_,'fro');
[p,n] = size(X);
tic;
[U,S,V] = svds(X,r);
Wtil = U*sqrt(S);
Htil = sqrt(S)*V';

if init == 0
    [W,H] = INIT_random(p,r,n);
```

```

% Subtractive clustering
elseif init == 1
    [W,H] = INIT_subclustering(X,r,n);

% NNDSVD
elseif init == 2
    [W,H] = INIT_NNDSVD(X,r);

% random Xcol
elseif init == 3
    [W,H] = INIT_Xcol(X,r,n);

% random-C
elseif init == 4
    [W,H] = INIT_randomC(X,r,n);

% co-occurrence
elseif init == 5
    [W,H] = INIT_cooccurrence(X,p,r,n);

% k-means with random column initialization
elseif init == 6
    [W,H] = INIT_kmeans(X,p,r,n,0,0,50);

% spherical k-means with random column initialization
elseif init == 7
    [W,H] = INIT_sphericalKmeans(X,p,r,n,0,0,50);

% PCA initilization
elseif init == 8
    [W,H] = INIT_PCA(X,r,n);

% SVD-NMF
elseif init == 9
    [W,H] = INIT_SVD_NMF(X,r);

% fuzzy c-means
elseif init == 10
    [W,H] = INIT_fuzzyMeans(X,p,r,n,0,0,50);

% NNDSVD with low-rank correction
elseif init == 11
    [W,H] = INIT_NNDSVD_LRC(X,r);

% given init
else
    W = W_init;
    H = H_init;

end

% properly scale W
alpha = (sum(sum(X*H'.*W)))/(sum(sum((W'*W).*(H*H'))));
W = alpha*W;

e = zeros(T,1);
t = 1;
%thresh = 0;
%norms = zeros(1,T);

% start timer
lr_time = 0;
time = zeros(1,T);
%first = 1;
%tic;
while (t <= T)
    % do not include time to calculate error in timing measurements
    pt = toc;
    e(t) = norm(X_ - W*H,'fro')/normX;
        %if (first == 1)
    %    time(t) = 0;
    %    first = 0;
    %else
```

```
        lr_time = lr_time + pt;
        time(t) = lr_time;
    %end

    %gradW = (W*(H*H') - X*H');
    %gradH = (W'*W)*H - W'*X;

    %if (t == 1)
    %    thresh = norm([gradW; gradH'],'fro');
        %fprintf('init grad norm %f\n', thresh);
    %    thresh = epsilon*thresh;
    %end

    % stopping condition
    %projnorm = norm([gradW(W>0); gradH(H>0)],'fro');
    %norms(t) = projnorm;
    %if (projnorm < thresh)
    %    %fprintf('init grad norm %f\n', projnorm);
    %    e(t+1:end) = e(t);
    %    break;
    %end
    tic;
    A = Wtil*(Htil*H');
    B = H*H';
    for l = 1:r
        s = zeros(p,1);
        for k = 1:r
            if k ~= l
                s = s + W(:,k)*(B(k,l));
            end
        end
        Z = (A(:,l) - s)/B(l,l);
        W(:,l) = max(Z, 0);
        if W(:,l) == 0
            W(:,l) = 1e-16*max(W(:));
        end
    end

    A = Htil'*(Wtil'*W);
    B = W'*W;
    for l = 1:r
        s = zeros(n,1);
        for k = 1:r
            if k ~= l
                s = s + H(k,:)'*(B(k,l));
            end
        end
        Z = (A(:,l) - s)/B(l,l);
        H(l,:) = max(Z, 0)';
        if isequal(H(l,:), zeros(size(H(l,:))))
            H(l,:) = 1e-16 + (H(l,:));
        end
    end
    t = t + 1;
end
```

```
function [W, H, time] = SPA(X, r)

tic;
[p,n] = size(X);
W = zeros(p,r);
R = X;
norms = zeros(1,n);
for i = 1:n
    norms(i) = norm(X(:,i))^2;
end

for i = 1:r
    [val,q] = max(norms);
    d = R(:,q)'*R/(val+10^-9);
    R = R - R(:,q)*d;
    norms = norms - d.^2*val;
    W(:,i) = X(:,q);
end
```

```

H = solveNNLS_BP(X,W,r,n,200);
H(isnan(H)) = 0;
H(isinf(H)) = 0;
time = toc;
```

```
function [W, H, time] = Hottopixx(X, r, T)

tic;
X_ = X;
[p,n] = size(X);
for i = 1:n
    X(:,i) = X(:,i)/norm(X(:,i));
end
q = randn(n,1);
C = zeros(n,n);
beta = 0;
sp = 10^-7;
sd = 1;
mu = zeros(n,1);
for i = 1:n
    mu(i) = length(find(abs(X(:,i))<=0.001))/p;
end
for t = 1:T
    for i = 1:p
        k = randi(p);
        C = C + sp*sign(X(k,:)' - C*X(k,:)')*X(k,:)-sp*diag(mu.*(beta*ones(n,1) - q));
    end
    for i = 1:n
        z = C(i,:);
        o = zeros(1,n);
        [z,inds] = sort(z,'descend');
        sigma = z(1);
        kc = 0;
        for k = 2:n
            if z(k) <= max(min(sigma,1),0)
                kc = k-1;
                break;
            else
                sigma = (k-1)*sigma/k + z(k)/k;
            end
        end
        o(1) = max(min(sigma,1),0);
        if kc == 0
            kc = n;
        end
        for k = 2:kc
            o(k) = max(min(sigma,1),0);
        end
        for k = kc+1:n
            o(k) = max(z(k),0);
        end
        C(i,inds) = o;
    end
    beta = beta + sd*(trace(C)-r);
end
d = diag(C);
[u,inds] = sort(d,'descend');
W = X_(:,inds(1:r));
H = solveNNLS_BP(X_,W,r,n,200);
time = toc;
```

```
function [W, H, time] = FastAnchorWords(X, r)

tic;
epsilon = 1/(sqrt(10));
[p,n] = size(X);
W = zeros(p,r);

D = eye(n);
for i = 1:n
    if rand > 0.5
        D(i,i) = -1;
    end
```

```
end
n_ = 2^(ceil(log(n)/log(2)));
H = hadamard(n_)/sqrt(n);
H = H(1:n,1:n);
k = round(4*log(n)/epsilon^2);
q = min(log(n)/(epsilon*n),1);
P = zeros(k,n);
for i = 1:k
    for j = 1:n
        if rand < q
            P(i,j) = randn()/sqrt(q);
        end
    end
end
R = P*H*D*X'*X;

norms = zeros(1,n);
for i = 1:n
    norms(i) = norm(R(:,i))^2;
end


K = zeros(1,r);
for i = 1:r
    [val,q] = max(norms);
    d = R(:,q)'*R/(val+10^-6);
    R = R - R(:,q)*d;
    norms = norms - d.^2*val;
    K(i) = q;
end

R=X;
norms_ = zeros(1,n);
for s = 1:n
    norms_(s) = norm(R(:,s))^2;
end
for i = 1:r
    norms = norms_;
    for j = 1:n
        proj = R(:,j);
        for l = 1:r
            if l == i
                continue;
            end
            val = norms_(K(l));
            d = (R(:,K(l))'*proj)/(val+10^-6);
            proj = proj - R(:,K(l))*d;
            norms(j) = norms(j) - d^2*val;
        end
    end

    % added to prevent columns of X from being the same
    [val,q] = max(norms);
    while (~isempty(find(K(1:i-1)==q,1)))
        norms(q) = 0;
        [val,q] = max(norms);
    end
    K(i) = q;
end
K;
W = X(:,K);
H = solveNNLS_BP(X,W,r,n,200);
H(isnan(H)) = 10^-6;
H = min(H,10^6);
if (norm(X-W*H,'fro')/norm(X,'fro') > 10)
alpha = (sum(sum(X*H'.*W)))/(sum(sum((W'*W).*(H*H'))));
W = alpha*W;
end
time = toc;
```

```
function [W, H, e, time] = TSVDNMF(X, r)

% http://proceedings.mlr.press/v48/bhattacharya16.pdf
tic;
```

```
[p,n] = size(X);
T = 50;
W = zeros(p,r);
H = zeros(r,n);

e0 = 0.04;
nu = 1.05;
alpha = 0.9;

D = tsvdnmf_thresh(X,alpha,e0);
%D = X;
[U,S,V] = svds(D,r);
Dr = U*S*V';

[W,init_py,time,e] = kmeans(Dr,p,r,n,T);
[W,py,time,e] = lloyds_kmeans(D,p,r,n,init_py,T);

% STEP 4
g = zeros(p,r);
place = floor(e0*n/2);
for i = 1:p
    for l = 1:r
        row_cluster = X(i,py==l);
        row_cluster = sort(row_cluster);
        if (length(row_cluster)-place + 1 > 0)
            g(i,l) = row_cluster(end-place+1);
        elseif isempty(row_cluster)
            g(i,l) = 0;
        end
    end
end

% J_l is the set of i's for which l is maximum in g(i,l)
J = zeros(r,p);
counts = zeros(r,1);
for i = 1:p
    [val1, ind1] = max(g(i,:));
    g(i,ind1) = intmin;
    [val2, ind2] = max(g(i,:));
    if (val1 > nu*val2)
        counts(ind1) = counts(ind1)+1;
        J(ind1,counts(ind1)) = i;
    end
end

cap = floor(e0*n/4);
% STEP 5
for l = 1:r
    sums = zeros(1,n);
    % if a basis vector has no dominant features,
    if (counts(l) == 0)
        if (~isempty(find(py==l,1)))
            W(:,l) = sum(X(:,py==l),2)/length(find(py==l));
        else
            W(:,l) = 0;
        end
        continue;
    end
    for j = 1:n
        s = 0;
        i = 1;
        while (i <= p && J(l,i) ~= 0)
            s = s + X(J(l,i),j);
            i = i+1;
        end
        sums(j) = s;
    end
    [sorted, inds] = sort(sums);
    if (counts(l) == 0)
        counts(l) = 1;
    end
    W(:,l) = sum(X(:,inds(end-cap+1)),2)/counts(l);
end

```

```
W(isnan(W)) = 0;
W(isinf(W)) = 0;
H = solveNNLS_BP(X,W,r,n,200);
H(isnan(H)) = 0;
H(isinf(H)) = 0;
time=toc;
e = norm(X-W*H,'fro');
```

```
function D = tsvdnmf_thresh(X, alpha, e0)

e4 = 10^-6;
[p,n] = size(X);
R = ones(p,1);
cX = X;

place = floor(e0*n/2);
% STEP 2
nus = zeros(p,1);
zetas = zeros(p,1);
C = zeros(p,n);
D = zeros(p,n);
for i = 1:p
    row = X(i,:);
    row = sort(row);
    nus(i) = row(end-place+1);
    zetas(i) = alpha*nus(i) - 2*e4;

    if zetas(i) >= 0
        C(i,:) = X(i,:) > zetas(i);
        D(i,:) = sqrt(zetas(i))*C(i,:);
    else
        R(i) = 0;
    end

end

mags = zeros(p,1);
for i = 1:p
    mags(i) = length(find(C(i,:)>0));
end

[sW, inds] = sort(mags);

for i = 1:p
    realI = inds(i);
    if (R(realI) == 0)
        continue;
    end
    for i_ = i+1:p
        realI_ = inds(i_);
        if (mags(realI) > mags(realI_) - e0*n/8 )
            continue;
        end
        if (length(find(C(realI,:)-C(realI_,:)>0)) <= e0*n/4)
            for j = 1:n
                if (C(realI_,j)==1 && C(realI,j)==0)
                    D(realI_,j) = 0;
                end
            end
            R(realI_) = 0;
        end
    end
end
```

```
function [W,py,time,e] = lloyds_kmeans(X,p,r,n,init_py,T)

normX = norm(X,'fro');

% use the given initialization

time = zeros(1,T);
e = zeros(1,T);

t = 1;
```

```
C = zeros(p,r);
d = zeros(n,r);
lr_time = 0;
tic;
while t <= T
    pt = toc;
    theta = 0;
    % compute distances
    for j = 1:r
        for i = 1:n
            d(i,j) = norm(X(:,i) - C(:,j));
        end
    end
    [tmp, py] = min(d,[],2);
    for j = 1:r
        inds = find(py==j);
        nj = length(inds);
        for i = 1:nj
            theta = theta + norm(X(:,inds(i)) - C(:,j))/norm(X(:,inds(i)));
        end
    end
    e(t) = theta;

    if (t == 1)
        time(t) = 0;
    else
        lr_time = lr_time + pt;
        time(t) = lr_time;
    end
    tic;

    if (t == 1)
        py = init_py;
    else
        % compute distances
        for j = 1:r
            for i = 1:n
                d(i,j) = norm(X(:,i) - C(:,j));
            end
        end
        % compute new clusters
        [tmp, py] = min(d,[],2);
    end

    % compute new centroids
    for j = 1:r
        s = zeros(p,1);
        inds = find(py==j);
        nj = length(inds);
        for i = 1:nj
            s = s + (X(:,inds(i)) - C(:,j));
        end
        if (nj ~= 0)
            s = s/nj;
        end
        C(:,j) = C(:,j) + s;
    end
    t = t + 1;
end

W = C;
```

```
function [W, H, time] = UoI(X,r,s)

tic;
[p,n] = size(X);
B1 = 20*s;
B2 = 10*s;
k = ceil(n/5);
k_ = ceil(n/5);
Wbig = zeros(p,B1*r);
S = zeros(B1,k);

for i = 1:B1
```

```
    s = datasample(1:n,k,'Replace',true);
    W = twoBlockCD_MU_KL(X(:,s),r,100,0,0,0,10^-6,X(:,s));
    S(i,:) = s';
    Wbig(:,r*(i-1)+1:r*i) = W;
end

% clustering
t = 0;
eps = p/16;
minPts = B1/8;
c = 0;
norms = zeros(B1*k,B1*k);
for i = 1:B1*r
    for j = 1:B1*r
        norms(i,j) = norm(Wbig(:,i) - Wbig(:,j));
    end
end
while ((t < 10 && c > 2*r)|| c < r)
    [labels,c] = DBSCAN_(Wbig, eps, minPts, norms);
    if c == r
        break;
    end
    numNoisy = length(find(labels==-1));
    if numNoisy == B1*r
        eps = 2.8*eps;
    elseif numNoisy == 0
        eps = eps/3.2;
    elseif numNoisy > B1*r/10
        eps = 1.2*eps;
    else
        eps = eps/1.5;
    end
    t = t+1;
    % if data is not conducive to density-based clustering, use kmeans
    % instead
    if t>30
        [dub,labels] = kmeans(Wbig,p,r,r*B1,50);
        c=r;
        break;
    end
end

pops = zeros(1,c);
for i = 1:c
    pops(i) = length(find(labels==i));
end

if c > r
    [pops,inds] = sort(pops);
    clusts = inds(end-r+1:end);
else
    clusts = 1:c;
end

find(labels==clusts(end))
W = zeros(p,r);
for i = 1:r
    W(:,i) = sum(Wbig(:,find(labels==clusts(i))),2)/pops(end-r+i);
end

Hbig = zeros(r,k*B1);
for i = 1:B1
    H = solveNNLS_BP(X(:,S(i,:)),W,r,k,50);
    Hbig(:,k*(i-1)+1:k*i) = H;
end

% build Hidx
Hidx = zeros(r,n);
for j = 1:n
    [row,col] = find(S ==j);
    qj = length(row);
    for l = 1:r
        bool = 1;
        for i = 1:qj
```

```
            if Hbig(l,k*(row(i)-1)+col(i)) < 0.000001
                bool = 0;
                break;
            end
        end
        if (bool)
            Hidx(l,j) = 1;
        end
    end
end

% weight estimation
Hbig = zeros(r,n*B2);
H = zeros(r,n);
for i = 1:B2
    s = datasample(1:n,k_,'Replace',true);
    H_ = zeros(r,n);
    for j = 1:k_
        inds = find(Hidx(:,s(j)) == 1);
        H_(inds,s(j)) = solveNNLS_BP(X(:,s(j)),W(:,inds),length(inds),1,20);
    end
    H = H + H_/B2;
end
% properly scale H
H(isnan(H)) = 0;
H(isinf(H)) = 0;
alpha = (sum(sum(X*H'.*W)))/(sum(sum((W'*W).*(H*H'))));
H = alpha*H;
time = toc;
% note: this yields a solution whose zero indices are exactly the zero
% indices of the H generated by solveNNLS_BPP for fixed W
```

```
function [labels,c] = DBSCAN(W, eps, minPts, norms)

[p,n] = size(W);
c = 0;
labels = zeros(1,n);

for i = 1:n
    if labels(i) ~= 0
        continue;
    end
    i;
    neighbors = [];
    count = 0;
    for j = 1:n
        if norms(i,j) < eps
            count = count +1;
            neighbors(count) = j;
        end
    end
    if (count < minPts)
        % label as noise
        'a';
        labels(i) = -1;
        continue;
    end
    % next cluster label
    c = c+1;
    labels(i) = c;
    z = 1;
    while z <= length(neighbors)
        z;
        if neighbors(z) == i
            z = z+1;
            'a';
            continue;
        end
        if labels(neighbors(z)) == -1
            z = z+1;
            labels(j) = c;
            'b';
            continue;
        end
        if labels(neighbors(z)) ~= 0
            z = z+1;
            'c';
            length(neighbors)
            continue;
        end

        labels(neighbors(z)) = c;
        neigh2 = [];
        count2=0;
        cur =0;
        for k = 1:n
            if norms(neighbors(z),k) < eps
                count2 = count2 +1;
                if (labels(k) <= 0 && isempty(find(neighbors == k,1)))
                    cur = cur+1;
                    neigh2(cur) = k;
                end
            end
        end
        if count2 >= minPts
            neighbors = [neighbors,neigh2];
        end
        z = z+1;
    end
end
```

```
function [] = TEST_last()

% Tests NMF algorithms over heavy-tailed data, noisy data, binary data,
% sparsity, dimension and debugging of algorithm execution on educational data
% Author: Liam Collins

%p = 361;
%r = 49;
%n = 2429;
%X = zeros(p,n);

epsilon = 0.0000001;

% construct X from CBLI face dataset
% used for debugging
for i = 1:0
    filename = strcat('./face/face', sprintf('%05d',i));
    filename = strcat(filename, '.pgm');
    A = imread(filename);
    X(:,i) = reshape(A,361,1);
    % normalize columns of X
    X(:,i) = X(:,i)/norm(X(:,i));
end

% all methods share same initialization
T = 400;  % max number of iterations
M = 3;  % number of trials
mm = 1; % number of variations of parameter to test

ps = [20, 100, 250, 500];
ns = ps;
ks = [5,10,20,40];
zetas = [0,1/4,1/2,1];

%figure(1)
%clf;
%figure(2)
%clf;
%figure(3)
%clf;
figure(22)
clf;

p = 220;
n = 256;
r = 16;
init = 12;
```

```

for j = 1:4

zeta = zetas(j);

avgTime_MU = zeros(1,100*T);
avgTime_BP = zeros(1,0.5*T);
avgTime_HALS = zeros(1,20*T);
avgTime_lra = zeros(1,20*T);
avgTime_ADMM = zeros(1,100*T);
avgTime_SPA = zeros(1,4);
avgTime_FAW = zeros(1,4);
avgTime_Hott = zeros(1,4);
avgTime_TSVD = zeros(1,4);
avgTime_UOI = zeros(1,4);

avgE_MU = zeros(100*T,1);
avgE_ANLS_BP = zeros(0.5*T,1);
avgE_HALS = zeros(20*T,1);
avgE_lra = zeros(20*T,1);
avgE_ADMM = zeros(100*T,1);
avgE_SPA = zeros(1,4);
avgE_FAW = zeros(1,4);
avgE_Hott = zeros(1,4);
avgE_TSVD = zeros(1,4);
avgE_UOI = zeros(1,4);

e_SPA = zeros(1,4);
e_UOI = zeros(1,4);
e_TSVD = zeros(1,4);
e_FAW = zeros(1,4);
e_Hott = zeros(1,4);

time_SPA = zeros(1,4);
time_UOI = zeros(1,4);
time_TSVD = zeros(1,4);
time_FAW = zeros(1,4);
time_Hott = zeros(1,4);

fin = 20;
time_SPA(4) = fin;
time_UOI(4) = fin;
time_TSVD(4) = fin;
time_FAW(4) = fin;
time_Hott(4) = fin;

e_SPA(1:2) = 100;
e_Hott(1:2) = 100;
e_UOI(1:2) = 100;
e_FAW(1:2) = 100;
e_TSVD(1:2) = 100;

for i = 1:M

    X = load('SwimmerDatabase');
    X = (X.M)';
    normX = norm(X,'fro');
    if j == 2
        X = X/normX;
        normX = 1;
        N = abs(randn(p,n));
        N = 0.25*N/norm(N,'fro');
        S = max(X+N,0);
    else
        S = X;
    end


    % TEST 1
    %X = abs(randn(p,n));
    %normX = norm(X, 'fro');
    %S=X;

    % TEST 2
    if (0)
    W = abs(randn(p,r));
    H = abs(randn(r,n));
    cols = datasample(1:n,r,'Replace',false);
    for s = 1:r
        H(:,cols(s)) = zeros(r,1);
        H(s,cols(s)) = 1;
    end

    X = W*H;
    normX = norm(X, 'fro');
    X = X/normX;
    normX = 1;
    N = abs(randn(p,n));
    N = zeta*N/norm(N,'fro');
    S = max(X+N,0);

    end

    if (0)
    % TEST 3
    if j == 1
        W = abs(randn(p,r));
        H = abs(randn(r,n));
        percSparse = 50;
        numInds = percSparse*p*r/100;
        data = 1:p*r;
        inds = datasample(data,numInds,'Replace',false);
        W(inds) = 0;
        numInds = percSparse*n*r/100;
        data = 1:n*r;
        inds = datasample(data,numInds,'Replace',false);
        H(inds) = 0;
        cols = datasample(1:n,r,'Replace',false);
        for s = 1:r
            H(:,cols(s)) = zeros(r,1);
            H(s,cols(s)) = 1;
        end
        X = W*H;
        normX = norm(X, 'fro');
        X = X/normX;
        normX = 1;
        N = abs(randn(p,n));
        N = 0.25*N/norm(N,'fro');
        S = max(X+N,0);
    elseif j == 2
        W = abs(randn(p,r));
        H = abs(randn(r,n));
        percSparse = 95;
        numInds = percSparse*p*r/100;
        data = 1:p*r;
        inds = datasample(data,numInds,'Replace',false);
        W(inds) = 0;
        numInds = percSparse*n*r/100;
        data = 1:n*r;
        inds = datasample(data,numInds,'Replace',false);
        H(inds) = 0;
        cols = datasample(1:n,r,'Replace',false);
        for s = 1:r
            H(:,cols(s)) = zeros(r,1);
            H(s,cols(s)) = 1;
        end
        X = W*H;
        normX = norm(X, 'fro');
        X = X/normX;
        normX = 1;
        N = abs(randn(p,n));
        N = 0.25*N/norm(N,'fro');
        S = max(X+N,0);

    elseif j == 3
        X = abs(randn(p,n));
        percSparse = 50;
        numInds = percSparse*p*n/100;
```

```
            data = 1:p*n;
            inds = datasample(data,numInds,'Replace',false);
            X(inds) = 0;
            normX = norm(X, 'fro');
            S=X;
        else
            X = abs(randn(p,n));
            percSparse = 95;
            numInds = percSparse*p*n/100;
            data = 1:p*n;
            inds = datasample(data,numInds,'Replace',false);
            X(inds) = 0;
            normX = norm(X, 'fro');
            S=X;
        end
        end

        if (0)
        % TEST 4
        if (j == 2)
            X = random('Generalized Pareto',1,1,1,[p,n]); %1-1 setup
            S =X;
            normX = norm(X, 'fro');
        elseif (j < 0)
            %X = random('Weibull',1,0.5,[p,n]);
        elseif (j < 0)
            %X = exp(randn(p,n));
        elseif (j == 1)
            %X = random('LogLogistic',1,1,[p,n]);
            W = random('Generalized Pareto',1,1,1,[p,r]);
            H = random('Generalized Pareto',1,1,1,[r,n]);
            cols = datasample(1:n,r,'Replace',false);
            for s = 1:r
                H(:,cols(s)) = zeros(r,1);
                H(s,cols(s)) = 1;
            end
            X = W*H;
            normX = norm(X, 'fro');
            X = X/normX;
            normX = 1;
            N = abs(randn(p,n));
            N = 0.25*N/norm(N,'fro');
            S = max(X+N,0);
        end
        end

        W_init = abs(randn(p,r));
        H_init = abs(randn(r,n));

        [W, H, time] = UoI(S,r,4);
        err_UOI = norm(X- W*H,'fro')/normX;
        e_UOI(3:4) = err_UOI;
        time_UOI(2) = time;
        time_UOI(3) = time+epsilon;

        [W,H,e_MU,iter_MU, time_MU] = twoBlockCD_MU(S,r,100*T,init,W_init,H_init,epsilon,X);

        [W,H,e_HALS,iter_HALS, time_HALS] = twoBlockCD_HALS(S,r,20*T,init,W_init,H_init,epsilon,X);

        [W,H,e_ANLS_BP,iter_ANLS_BP, time_BP] = twoBlockCD_ANLS(S,r,0.5*T,50,init,W_init,H_init,
            epsilon,3,X);

        [Wp, Hp, e_ADMM, time_ADMM] = ADMM(S,r,100*T,init,W_init,H_init,epsilon,X);

        [W, H, e_lra, time_lra] = lraNMF_HALS(S,r,20*T,init,W_init,H_init,epsilon,X);

        [W, H, time] = SPA(S,r);
        err_SPA = norm(X- W*H,'fro')/normX;
        e_SPA(3:4) = err_SPA;
        time_SPA(2) = time;
        time_SPA(3) = time+epsilon;

        [W, H, time] = Hottopixx(S,r,T/4);
        err_Hott = norm(X- W*H,'fro')/normX;
```

```
    e_Hott(3:4) = err_Hott;
    time_Hott(2) = time;
    time_Hott(3) = time+epsilon;

    [W, H, e, time] = TSVDNMF(S,r);
    err_TSVD = e/normX;
    e_TSVD(3:4) = err_TSVD;
    time_TSVD(2) = time;
    time_TSVD(3) = time+epsilon;

    [W, H, time] = FastAnchorWords(S,r);
    err_FAW = norm(X- W*H,'fro')/normX;
    e_FAW(3:4) = err_FAW;
    time_FAW(2) = time;
    time_FAW(3) = time+epsilon;

    avgTime_MU = avgTime_MU + time_MU/M;
    avgTime_ADMM = avgTime_ADMM + time_ADMM/M;
    avgTime_HALS = avgTime_HALS + time_HALS/M;
    avgTime_BP = avgTime_BP + time_BP/M;
    avgTime_lra = avgTime_lra + time_lra/M;
    avgTime_SPA = avgTime_SPA + time_SPA/M;
    avgTime_UOI = avgTime_UOI + time_UOI/M;
    avgTime_FAW = avgTime_FAW + time_FAW/M;
    avgTime_Hott = avgTime_Hott + time_Hott/M;
    avgTime_TSVD = avgTime_TSVD + time_TSVD/M;

    avgE_MU = avgE_MU + e_MU/M;
    avgE_ANLS_BP = avgE_ANLS_BP + e_ANLS_BP/M;
    avgE_ADMM = avgE_ADMM + e_ADMM/M;
    avgE_HALS = avgE_HALS + e_HALS/M;
    avgE_lra = avgE_lra + e_lra/M;
    avgE_SPA = avgE_SPA + e_SPA/M;
    avgE_UOI = avgE_UOI + e_UOI/M;
    avgE_FAW = avgE_FAW + e_FAW/M;
    avgE_TSVD = avgE_TSVD + e_TSVD/M;
    avgE_Hott = avgE_Hott + e_Hott/M;


end


figure(22)
subplot(2,2,j)
semilogy(avgTime_MU, avgE_MU);
hold on
semilogy(avgTime_BP, avgE_ANLS_BP);
hold on
semilogy(avgTime_HALS, avgE_HALS);
hold on
semilogy(avgTime_ADMM, avgE_ADMM);
hold on
semilogy(avgTime_lra, avgE_lra);
hold on
semilogy(avgTime_SPA, avgE_SPA);
hold on
semilogy(avgTime_FAW, avgE_FAW);
hold on
semilogy(avgTime_Hott, avgE_Hott,'--');
hold on
semilogy(avgTime_TSVD, avgE_TSVD,'--');
hold on
semilogy(avgTime_UOI, avgE_UOI,'--');
hold on
legend('MU','ANLS BPP','HALS','ADMM','lraNMF-HALS','SPA','FastAnchorWords','Hottopixx','TSVDNMF','
    UoI Clustering');
xlabel('Time (s)');
ylabel('||X - WH||_F/||X||_F');

end
```

```
p = 40;
r = 8;
n = 24;
```

```
B = 20;

data = 1:(r*n);
W = 2*abs(randn(p,r));
%W = zeros(p,r);
%for j = 1:r
%    W(5*(j-1)+1:5*j,j) = rand(5,1);
%end
cond(W)
H = abs(randn(r,n));
percSparse = 75;
numInds = percSparse*r*n/100;
inds = datasample(data,numInds,'Replace',false);
H(inds) = 0;

N = abs(randn(p,n));
N = rand(p,n);
for i = 1:r
    W(:,i) = W(:,i)/norm(W(:,i));
end
for i = 1:n
    N(:,i) = N(:,i)/norm(N(:,i));
end
X = W*H + N;

data = 1:n;
q = zeros(1,B*r);
m = zeros(1,B*r);
for i = 1:B
    inds = datasample(data,2*r,'Replace',true);
    W_ = twoBlockCD_MU_KL(X(:,inds),r,100,0,0,0,10^-6,X(:,inds));
    for k = 1:r
        W_(:,k) = W_(:,k)/norm(W_(:,k));
    end
    for j = 1:r
        q(j + (i-1)*r) = max(W'*W_(:,j));
        m(j + (i-1)*r) = max(N'*W_(:,j));
    end
end
figure(55)
clf;
scatter(1:r*B,q,'b');
hold on;
scatter(1:r*B,m,'r');

figure(56)
plot(abs(q-m));
```

Bibliography

[1] N. Ailon and B. Chazelle. Approximate nearest neighbors and the fast johnson-lindenstrauss transform. 2006.

[2] U. Araújo, B. Saldanha, R. Galvão, T. Yoneyama, H. Chame, and V. Visani. The successive projections algorithm for variable selection in spectroscopic multicomponent analysis. *Chemometrics and Intelligent Laboratory Systems*, 57(2):65–73, 2001.

[3] S. Arora, R. Ge, Y. Halpern, D. Mimno, A. Moitra, D. Sontag, Y. Wu, and M. Zhu. A practical algorithm for topic modeling with provable guarantees. volume 28, pages 280–288, 2013.

[4] S. Arora, R. Ge, R. Kannan, and A. Moitra. Computing a nonnegative matrix factorization – provably. *Proceeding of the 44th Symp. on Theory of Computing*, page 145–162, 2012.

[5] R. S. Baker, A. T. Corbett, and V. Aleven. More accurate student modeling through contextual estimation of slip and guess probabilities in bayesian knowledge tracing. volume 5091, page 406–415, 2008.

[6] M. H. Van Benthem and M. R. Keenan. Fast algorithm for the solution of large-scale nonnegativity-constrained least squares problems. *Journal of Chemometrics*, 18:441–450, 2004.

[7] M. Berry, M. Browne, A. Langville, V. Pauca, and R. Plemmons. Algorithms and applications for approximate nonnegative matrix factorization. *Computational Statistics & Data Analysis*, 52(1):155–173, 2007.

[8] D. Bertsekas. On the goldstein-levitin-polyak gradient projection method. *IEEE Transactions on Automatic Control*, 21:174–184, 1976.

[9] D. Bertsekas. *Nonlinear Programming*, volume second edition. Athena Scientific, Belmont, MA, 1999.

[10] J.C. Bezdek. *Pattern recognition with fuzzy objective functions algorithms.* Plenum, 1981.

[11] C. Bhattacharya, N. Goyal, R. Kannan, and J. Pani. Non-negative matrix factorization under heavy noise. Jun 2016.

[12] V. Bittorf, B. Recht, C. Re, and J. Tropp. Factoring nonnegative matrices with linear programs. volume 25, 2012.

[13] L. Bottou and Y. Bengio. Convergence properties of the k-means algorithms. volume 7, pages 585–592, 1995.

[14] K. E. Bouchard, A. F. Bujan, F. Roosta-Khorasani, S. Ubaru, Prabhat, A. M. Snijders, J.-H. Mao, E. F. Chang, M. W. Mahoney, and S. Bhattacharyya. Union of intersections (uoi) method for interpretable data driven discovery and prediction. 2017.

[15] N. Boumal. Mat 321 numerical methods lecture notes. 2018.

[16] C. Boutsidis and E. Gallopoulos. Svd based initialization: A head start for nonnegative matrix factorization. *Pattern Recognition*, 41(4):1350–1362, 2008.

[17] S. Boyd, N. Parikh, E. Chu, B. Paleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2011.

[18] C. Brinton and M. Chiang. Networks illustrated: Principles without calculus. `https://www.coursera.org/learn/networks-illustrated/home`, 2018.

[19] J.P. Brunet, P. Tamayo, T.R. Golub, and J.P. Mesirov. Metagenes and molecular pattern discovery using matrix factorization. *Proceedings of the National Academy of Science*, 101(12):4164–4169, 2004.

[20] P.H. Calamai and J.J. More. Projected gradient methods for linearly constrained problems. *Mathematical Programming*, 39:93–116, 1987.

[21] G. Casalino, N. Del Buono, and C. Mencar. Subtractive clustering for seeding non-negative matrix factorizations. *Information Sciences*, 257:369–387, 2013.

[22] S. Chiu. Fuzzy model identification based on cluster identification. *Journal of Intelligent Fuzzy Systems*, 2:267–278, 1994.

[23] A. Cichocki, R. Zdunek, A. H. Phan, and S. Amari. *Nonnegative Matrix and Tensor Factorizations: Applications to Exploratory Multi-way Data Analysis and Blind Source Separation*. John Wiley & Sons, Ltd, 2009.

[24] A. Cichoki, R. Zdunek, and S.I. Amari. *Hierarchical ALS Algorithms for Nonnegative Matrix and 3D Tensor Factorization. In: Lecture Notes in Computer Science*, volume 4666. Springer, 2007.

[25] M. Desmarais. Projected gradient methods for linearly constrained problems. *Proceedings of the 4th International Conference on Educational Data Mining*, pages 41–50, 2011.

[26] I. S. Dhillon, Y. Guan, and J. Kogan. Iterative clustering of high dimensional text data augmented by local search. 2002.

[27] I. S. Dhillon and D. S. Modha. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. *Machine Learning*, 42(1-2):143–175, 2001.

[28] D. Donoho and V. Stodden. When does non-negative matrix factorization give a correct decomposition into parts? 2003.

[29] A. Elazab, Y. Abdulazeem, S. Wu, and Q. Hu. An efficient initialization method for nonnegative matrix factorization. *Journal of Applied Sciences*, 11:354–359, 2011.

[30] A. Elazab, Y. Abdulazeem, S. Wu, and Q. Hu. Robust kernelized local information fuzzy c-means clustering for brain magnetic resonance image segmentation. *Journal of X-Ray Science and Technology*, 24:489–507, 2016.

[31] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. page 226–231. KDD'96, AAAI Press, 1996.

[32] S. Fiorini, V. Kaibel, K. Pashkovich, and D. Theis. Combinatorial bounds on nonnegative rank and extended formulations. *Discrete Mathematics*, 313(1):67–83, 2013.

[33] R. Ge, C. Jin, and Y. Zheng. No spurious local minima in nonconvex low rank problems: A unified geometric analysis. *Proceedings of the 34th International Conference on Machine Learning*, page 1233–1242, 2017.

[34] R. Ge and J. Zhou. Intersecting faces: Nonnegative matrix factorization with new guarantees. In *ICML*, page 2295–2303, 2015.

[35] N. Gillis. *Nonnegative matrix factorization: Complexity, algorithms and applications.* PhD thesis, UCLouvain, 2011.

[36] N. Gillis. Successive nonnegative projection algorithm for robust nonnegative blind source separation. *SIAM Journal Imaging Sciences*, 7(2):1420–1450, 2014.

[37] N. Gillis. The why and how of nonnegative matrix factorization. *arXiv:1401.5226*, 2014.

[38] N. Gillis and F. Glineur. Accelerated multiplicative updates and hierarchical als algorithms for nonnegative matrix factorization. *Neural Computation*, 24(4):1085–1105, 2012.

[39] N. Gillis and F. Glineur. On the geometric interpretation of the nonnegative rank. *Linear Algebra and its Applications*, 437(11):2685–2712, 2012.

[40] N. Gillis and R. Luce. Robust near-separable nonnegative matrix factorization using linear optimization. *The Journal of Machine Learning Research*, 15(1):1249–1280, 2014.

[41] N. Gillis and R. Luce. Faster matrix completion using randomized svd. *The Journal of Machine Learning Research*, 2018.

[42] N. Gillis and W.K. Ma. Enhancing pure-pixel identification performance via preconditioning. Jun 2014.

[43] N. Gillis and S. A. Vavasis. Fast and robust recursive algorithmsfor separable nonnegative matrix factorization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 698 – 714, 2014.

[44] N. Gillis and S. A. Vavasis. Semidefinite programming based preconditioning for more robust near-separable nonnegative matrix factorization. *SIAM Journal of Optimization*, 25(1):677–698, 2015.

[45] N. Gillis and S. A. Vavasis. Faster matrix completion using randomized svd. *The Journal of Machine Learning Research*, 2018.

[46] L. Grippo and M. Sciandrone. On the convergence of the block nonlinear gauss-seidel method under convex constraints. *Oper. Res. Lett.*, 26:127–136, 2000.

[47] N. D. Ho. *Nonnegative matrix factorization - algorithms and applications*. PhD thesis, UCLouvain, 2008.

[48] T. Hoffman. Probabilistic latent semantic indexing. In *Proc. 22nd International Conference on Research and Development in Information Retrieval (SIGIR)*, 1999.

[49] C.J. Hsieh and I. Dhillon. Fast coordinate descent methods with variable selection for non-negative matrix factorization. pages 1064–1072, Jun 2011.

[50] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown. Algorithm runtime prediction: The state of the art. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[51] A. Janecek and Y. Tan. Using population based algorithms for initializing nonnegative matrix factorization. pages 307–316, 2011.

[52] J. Kim and H. Park. Nonnegative matrix factorization based on alternating nonnegativity-constrained least squares and the active set method. *SIAM Journal on Matrix Analysis and Applications*, 30(2):713–730, 2008.

[53] J. Kim and H. Park. Toward faster nonnegative matrix factorization: A new algorithm and comparisons. pages 353–362, 2008.

[54] J. Kim and H. Park. Fast nonnegative matrix factorization: An active-set-like method and comparisons. *SIAM Journal on Scientific Computing (SISC)*, 33(6):3261–3281, 2011.

[55] D. Kitamura and N. Ono. Efficient initialization for nonnegative matrix factorization based on nonnegative independent component analysis.

[56] A. Kumar, V. Sindhwani, and P. Kambadur. Fast conical hull algorithms for near-separable non-negative matrix factorization. volume 28, page 231–239, 2013.

[57] A. Langville, C. D. Meyer, and R. Albright. Initializations for the nonnegative matrix factorization. In *Proc. 12th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining*, 2006.

[58] A. Langville, C. D. Meyer, R. Albright, J. Cox, and D. Duling. Algorithms, initializations, and convergence for the nonnegative matrix factorization. *Information Sciences*, 2006.

[59] D. Lee and H. Seung. Learning the parts of objects by nonnegative matrix factorization. *Nature*, 401:788–791, 1999.

[60] C.J. Lin. Projected gradient methods for nonnegative matrix factorization. *Neural Computation*, 19:2756–2779, 2007.

[61] T. Mizutani. Robustness analysis of preconditioned successive projection algorithm for general form of separable nmf problem. *Linear Algebra Applications*, 1-22:497, 2016.

[62] A. Moitra. An almost optimal algorithm for computing nonnegative rank. In *Proc. of the 24th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA '13)*, page 1454–1464, 2013.

[63] J. Nascimento and J Bioucas-Dias. Vertex component analysis: a fast algorithm to unmix hyperspectral data. *IEEE Trans. Geosci. Remote Sens.*, 43(4):898–910, 2005.

[64] V. Perrone, P. A. Jenkins, D. Spano, and Y. W. Teh. Poisson random fields for dynamic feature models, 2016.

[65] S.J. Prajapati and K.R. Jadhav. Brain tumor detection by various image segmentation techniques with introduction to non negative matrix factorization. *Brain*, 4(3):600–603, 2015.

[66] H. Qiao. New svd based initialization strategy for non-negative matrix factorization. *Pattern Recognition Letters*, 63:71–77, 2015.

[67] S. Sra. Lecture notes on "alternating minimization (and friends)", lecture 7, course 6.883, 2016.

[68] D. Sun and C. Févotte. Alternating direction method of multipliers for non-negative matrix factorization with the beta-divergence.

[69] A.M. Syed, S. Qazi, and N. Gillis. Improved svd-based initialization for nonnegative matrix factorization using low-rank correction. 2018.

[70] V. Tan and C. Fevotte. Automatic relevance determination in nonnegative matrix factorization. *Signal Processing with Adaptive Sparse Structured Representations*, 2009.

[71] S. Ubaru, K. Wu, and K. E. Bouchard. Uoi-nmf cluster: A robust nonnegative matrix factorization algorithm for improved parts-based decomposition and reconstruction of noisy data. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 241–248, Dec 2017.

[72] S. Vavasis. On the complexity of nonnegative matrix factorization. *SIAM Journal on Optimization*, pages 1364–1377, 2009.

[73] H. Wang, D. Li, and X. He. Estimation of high conditional quantiles for heavy-tailed distributions. *Journal of the American Statistical Association*, 107(2):1453–1464, 2012.

[74] X. Wang, X. Xie, and L. Lu. An effective initialization for orthogonal nonnegative matrix factorization. *Journal of Computational Mathematics*, 30:34–46, 2012.

[75] Y. X. Wang and Y. J. Zhang. Nonnegative matrix factorization: A comprehensive review. *IEEE Transactions on Knowledge and Data Engineering*, 25(6), 2012.

[76] S. Wild. *Seeding Non-Negative Matrix Factorizations with the Spherical K-Means Clustering*. PhD thesis, University of Colorado, 2002.

[77] T. Winters, C. Shelton, T. Payne, and G. Mei. Topic extraction from item level grades. *American Association for Artificial Intelligence 2005 Workshop on Educational Datamining*, 2005.

[78] S. Wu, A. Joseph, A. S. Hammonds, S. E. Celniker, B. Yu, and E. Frise. Stability-driven nonnegative matrix factorization to interpret spatial gene expression and build local gene networks. *Proceedings of the National Academy of Sciences*, 113(16):4290–4295, 2016.

[79] Y. Xu, W. Yin, Z. Wen, B. Paleato, and Y. Zhang. An alternating direction algorithm for matrix completion with nonnegative factors. *Frontiers of Mathematics in China*, 7(2):365–384, 2012.

[80] M. Yannakakis. Expressing combinatorial optimization problems by linear programs. *JCSS*, page 441–466, 1991.

[81] H.F. Yu, C.J. Hsieh, S. Si, and I. S. Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. page 765–774, 2012.

[82] L. Zhao, G. Zhuang, and X. Xu. Facial expression recognition based on pca and nmf. 2008.

[83] Z. Zheng, J. Yang, and Y. Zhu. Initialization enhancer for non-negative matrix factorization. *Journal of Engineering Applications of Artificial Intelligence*, 20:101–110, 2007.

[84] G. Zhou and A. Cichocki. Fast nonnegative matrix/tensor factorization based on low-rank approximation. *IEEE Transactions on Signal Processing*, pages 2928 – 2940, 2012.