CS168: The Modern Algorithmic Toolbox
Lecture #18: Linear and Convex Programming, with Applications to Sparse Recovery

Tim Roughgarden & Gregory Valiant*

May 25, 2016

1 The Story So Far

Recall the setup in compressive sensing. There is an unknown signal $\mathbf{z} \in \mathbb{R}^n$, and we can only glean information about $\mathbf{z}$ through linear measurements. We choose m linear measurements $\mathbf{a}_1, \ldots, \mathbf{a}_m \in \mathbb{R}^n$. "Nature" then chooses a signal $\mathbf{z}$, and we receive the results $b_1 = \langle \mathbf{a}_1, \mathbf{z} \rangle, \ldots, b_m = \langle \mathbf{a}_m, \mathbf{z} \rangle$ of our measurements, when applied to $\mathbf{z}$. The goal is then to recover $\mathbf{z}$ from $\mathbf{b}$.

Last lecture culminated in the following sparse recovery guarantee for compressive sensing.

Theorem 1.1 (Main Theorem) *Fix a signal length n and a sparsity level k. Let $\mathbf{A}$ be an $m \times n$ matrix with $m = \Theta(k \log \frac{n}{k})$ rows, with each of its mn entries chosen independently from the standard Gaussian distribution. With high probability over the choice of $\mathbf{A}$, for every k-sparse signal $\mathbf{z}$,[1] the unique optimal solution to the ℓ_1-minimization problem*

$$\min \|\mathbf{x}\|_1 \tag{1}$$

subject to

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{2}$$

is $\mathbf{z}$.

Theorem 1.1 is really kind of shocking.[2] We're solving the wrong optimization problem — ℓ_1-minimization rather than ℓ_0-minimization, which is what we really want (but NP-hard)

[1] Recall $\mathbf{z}$ is *k-sparse* if it has at most k non-zeroes.

[2] The proof is not easy, and is beyond the scope of CS168. Our focus is on interpretations, applications, and algorithms.

— and yet obtaining the right answer! How would one ever suspect that this could be true? Perhaps unsurprisingly, this unreasonable exact recovery was first observed empirically — by geophysicists in the late 1970s and early 1980s, and again in the early 21st century in the context of medical imaging. Only then, in the last decade or so, was theory developed to explain these empirical observations (beginning with [3, 6]).

Last lecture, we discussed why minimizing the ℓ_1 norm of a feasible solution promotes sparse solutions. Geometrically, the ℓ_1 ball is the "longest" along the standard basis vectors, and hence if we blow up a balloon centered at the origin in the shape of the ℓ_1 ball, its first point of contact with a subspace of linear system solutions (an affine subspace) tends to be relatively sparse, compared to other norms. Next, we discuss how to solve the ℓ_1-minimization problem efficiently using linear programming.

2 Linear Programming

2.1 Context

The more general a problem, the more computationally difficult it is. For example, sufficient generalization of a polynomial-time solvable problem often yields an NP-hard problem. If you only remember one thing about linear programming, make it this: *linear programming is a remarkable sweet spot balancing generality and computational tractability*, arguably more so than any other problem in the entire computational landscape.

Zillions of problems, including ℓ_1-minimization, reduce to linear programming. It would take an entire course to cover even just its most famous applications. Some of these applications are conceptually a bit boring but still very important — as early as the 1940s, the military was using linear programming to figure out the most efficient way to ship supplies from factories to where they were needed. Central problems in computer science that reduce to linear programming include maximum flow and bipartite matching. (There are also specialized algorithms for these two problems, see CS261.) Linear programming is also useful for NP-hard problems, for which it serves as a powerful subroutine in the design of heuristics (again, see CS261).

Despite this generality, linear programs can be solved efficiently, both in theory (meaning in worst-case polynomial time) and in practice (with input sizes up into the millions).

2.2 Using Linear Programming

You can think of linear programming as a restricted programming language for encoding computational problems. The language is flexible, and sometimes figuring out the right way to use it requires some ingenuity (as we'll see).

At a high level, the description of a linear program specifies what's allowed, and what you want. Here are the ingredients:

1. *Decision variables.* These are real-valued variables $x_1, \ldots, x_n \in \mathbb{R}$. They are "free," in the sense that it is the job of the linear programming solver to figure out the best

joint values for these variables.

2. *Constraints.* Each constraint should be linear, meaning it should have the form

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i$$

or

$$\sum_{j=1}^{n} a_{ij} x_j = b_i.$$

We didn't bother including constraints of the form $\sum_{j=1}^{n} a_{ij} x_j \geq b_i$, since these are equivalent to $\sum_{j=1}^{n} (-a_{ij}) x_j \leq -b_i$. All of the a_{ij}'s and b_i's are real-valued constants, meaning specific numbers (1, -5, 10, etc.) that are hard-coded into the linear program.

3. *Objective function.* Again, this should be linear, of the form

$$\min \sum_{j=1}^{n} c_j x_j.$$

It's fine to maximize instead of minimize: after all, $\max \sum_{j=1}^{n} c_j x_j$ yields the same result as $\min \sum_{j=1}^{n} (-c_j) x_j$.

So what's not allowed in a linear program? Terms like x_j^2, $x_j x_k$, $\log(1 + x_j)$, etc. So whenever a decision variable appears in an expression, it is alone, possibly multiplied by a constant. These linearity requirements may seem restrictive, but many real-world problems are well approximated by linear programs.

2.3 A Simple Example

To make linear programs more concrete and develop your intuition about them, let's look at a simple example. Suppose there are two decision variables x_1 and x_2 — so we can visualize solutions as points (x_1, x_2) in the plane. See Figure 1. Let's consider the (linear) objective function of maximizing the sum of the decision variables:

$$\max x_1 + x_2. \tag{3}$$

We'll look at four (linear) constraints:

$$x_1 \geq 0 \tag{4}$$
$$x_2 \geq 0 \tag{5}$$
$$2x_1 + x_2 \leq 1 \tag{6}$$
$$x_1 + 2x_2 \leq 1. \tag{7}$$

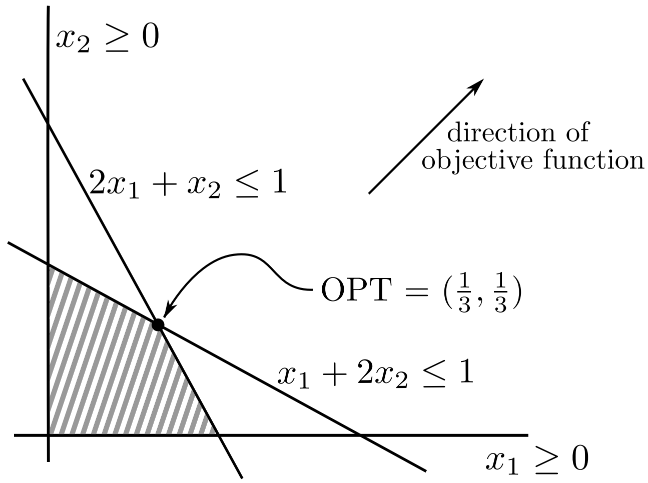

Figure 1: A linear program in 2 dimensions.

The first two inequalities restrict feasible solutions to the non-negative quadrant of the plane. The second two inequalities further restrict feasible solutions to lie in the shaded region depicted in Figure 1. Geometrically, the objective function asks for the feasible point furthest in the direction of the coefficient vector $(1, 1)$ — the "most northeastern" feasible point. Eyeballing the feasible region, this point is $(\frac{1}{3}, \frac{1}{3})$, for an optimal objective function value of $\frac{2}{3}$.

2.4 Geometric Intuition

This geometric picture remains valid for general linear programs, with an arbitrary number of dimensions and constraints: *the objective function gives the optimization direction, and the goal is to find the feasible point that is furthest in this direction.* Moreover, the feasible region of a linear program is just a higher-dimensional analog of a polygon.[3]

2.5 Algorithms for Linear Programming

Linear programs are not difficult to solve in two dimensions — for example, one can just check all of the vertices (i.e., "corners") of the feasible region. In high dimensions, linear programs are not so easy; the number of vertices can grow exponentially with the number of dimensions (e.g., think about hypercubes), so there's no time to check them all. Nevertheless, we have the following important fact.

Fact 2.1 *Linear programs can be solved efficiently.*

[3]Called a "polyhedron;" in the common special case where the feasible region is bounded, it is called a "polytope."

The theoretical version of Fact 2.1 states that there is a polynomial-time algorithm for linear programming.[4] The practical version of Fact 2.1 is that there are excellent commercial codes available for solving linear programs.[5] These codes routinely solve linear programs with millions of variables and constraints. One thing to remember about linear programming is that, for over 60 years, many people with significant resources — ranging from the military to large companies — have had strong incentives to develop good codes for it. This is one of the reasons that the best codes are so fast and robust.

In CS168, we won't discuss how the various algorithms for linear programming work.[6] While the key conceptual ideas are pretty natural, lots of details are required. Most professional researchers and optimizers treat linear programming as a "black box" — a subroutine that can be invoked at will, without knowledge of its inner details. We'll adopt this perspective as well in lecture and on Homework #9.

3 Linear Programming and ℓ_1-Minimization

We now show that the ℓ_1-minimization problem in Theorem 1.1 can be solved using linear programming. The only non-trivial issue is the objective function

$$\min \|\mathbf{x}\|_1 = \sum_{j=1}^{n} |x_j|,$$

which, because of the absolute values, is non-linear.

As a warm-up, suppose first that we know that the unknown signal $\mathbf{z}$ is component-wise non-negative (in addition to being k-sparse). Then, the ℓ_1-minimization problem is just

$$\min \sum_{j=1} x_j$$

subject to

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{8}$$

and

$$\mathbf{x} \geq 0. \tag{9}$$

The objective function is clearly linear. The n non-negativity constraints in (9) — each of the form $x_j \geq 0$ for some j — are linear. Each of the m equality constraints (8) has the form $\sum_{j=1}^{n} a_{ij}x_j = b_i$, and is therefore linear. Thus, this is a linear program.

We know the unknown signal $\mathbf{z}$ satisfies $\mathbf{A}\mathbf{x} = \mathbf{b}$ (by the definition of $\mathbf{b}$). We're also assuming that $\mathbf{z} \geq 0$. Hence, $\mathbf{z}$ is a feasible solution to the linear program. Since $\mathbf{x} \geq 0$ for

[4]The earliest, from 1979, is the "ellipsoid method" [7]; this was a big enough deal at the time that it made the New York Times [1].

[5]The open-source solvers are not as good, unfortunately, but are still useful for solving reasonably large linear programs (see Homework #9).

[6]Kudos to the reader who is bothered by this: linear programming is a beautiful subject, and we strongly encourage the reader to take a class or read a book (like [8]) on the subject.

every feasible solution, the objective function value $\sum_{j=1}^{n} x_j$ equals $\|\mathbf{x}\|_1$ for every feasible solution. We conclude that this linear program is a faithful encoding of ℓ_1-minimization for non-negative signals.

For the general case of real-valued signals $\mathbf{z}$, the key trick is to add additional variables that allow us to "linearize" the non-linear objective function in (1). In addition to the previous decision variables $x_1, \ldots, x_n$, our linear program will include auxiliary decision variables $y_1, \ldots, y_n$. The intent is for y_j to represent $|x_j|$. We use the objective function

$$\min \sum_{j=1}^{n} y_j, \tag{10}$$

which is clearly linear. We also add $2n$ linear inequalities, of the form

$$y_j - x_j \geq 0 \tag{11}$$

and

$$y_j + x_j \geq 0 \tag{12}$$

for every $j = 1, 2, \ldots, n$. Finally, we have the usual m linear consistency constraints

$$\mathbf{A}\mathbf{x} = \mathbf{b}. \tag{13}$$

Every feasible solution of this linear program satisfies all of the constraints, and in particular (11) and (12) imply that $y_j \geq \max\{x_j, -x_j\} = |x_j|$ for every $j = 1, 2, \ldots, n$. Observe further that at an optimal solution, equality must hold for every j: given a feasible solution with $y_j > x_j$ and $y_j > -x_j$ for some j, one can decrease y_j slightly to produce a new solution that is still feasible and that has slightly better (i.e., smaller) objective function value (10). It follows that the values of the variables $\mathbf{x}$ in an optimal solution to the linear program given by (10)–(13) is the optimal solution to the ℓ_1-minimization problem given in (1)–(2).

To further showcase the power and flexibility of linear programming, suppose that the results of the linear measurements are corrupted by noise. Concretely, assume that instead of receiving $b_i = \langle \mathbf{a}_i, \mathbf{z} \rangle$ for each measurement $i = 1, 2, \ldots, m$, we receive a value $b_i \in [\langle \mathbf{a}_i, \mathbf{z} \rangle - \epsilon, \langle \mathbf{a}_i, \mathbf{z} \rangle + \epsilon]$, where $\epsilon > 0$ is a bound on the magnitude of the noise. Now, the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ might well be infeasible — $\mathbf{z}$ is now only an approximately feasible solution. The linear program (10)–(13) is easily modified to accommodate noise — just replace the equality constraints (13) by two sets of inequality constraints,

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i + \epsilon$$

and

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i - \epsilon$$

for each $i = 1, 2, \ldots, m$. The guarantee in Theorem 1.1 can also be extended, with significant work, to handle noise [4].

Figure 2: Examples of convex and non-convex sets.

Remark 3.1 (Keep Linear Programming in Your Toolbox) This concludes our brief discussion of linear programming. While compressive sensing is a convenient excuse to discuss this powerful tool, don't forget that linear programming is useful for solving or approximating a huge range of applications drawn from many different domains. It's quite likely that one or more problems arising in your future work will be solvable using linear programming. The classic book [5] remains an excellent introduction to some of the applications.

4 Beyond Linear Programs: Convexity

We next discuss a generalization of linear programming that captures still more applications, without sacrificing too much computational efficiency. After describing this generalization, we give in Section 5 a representative application, the matrix completion problem.

4.1 Convex Sets

In Week 3 we mentioned that a good rule of thumb is to equate "convex" with "nice" and "non-convex" with "nasty," especially when optimization is concerned. Back then, we were studying gradient descent, which has much nicer properties for minimizing convex functions than for non-convex functions (in the latter case, the ending point depends on the starting point and might only be a local minimum). Here, convexity is in large part what's driving the computational tractability of linear programming.

Convexity is relevant for both sets and for functions. Intuitively, a subset $C \subseteq \mathbb{R}^n$ is convex if it is "filled in," meaning that it contains all line segments between its points. See Figure 2 for examples. Formally, C is *convex* if for every $\mathbf{x}, \mathbf{y} \in C$ and $\lambda \in [0,1]$, $\lambda\mathbf{x} + (1-\lambda)\mathbf{y} \in C$. (As λ ranges from 0 to 1, it traces out the line segment from $\mathbf{y}$ to $\mathbf{x}$.)

For example, the feasible region of every linear program is convex. To see this, first suppose there is only one constraint, which is an inequality. Then the feasible region is just a half-space, which is clearly convex. The feasible region of a linear program is an intersection of such half-spaces. (Note that an equality constraint is equivalent to the combination of two inequality constraints.) The intersection of convex sets C_1, C_2 is again convex — if $\mathbf{x}$ and $\mathbf{y}$ are in both C_1 and C_2, then the line segment between $\mathbf{x}$ and $\mathbf{y}$ lies inside both C_1 and

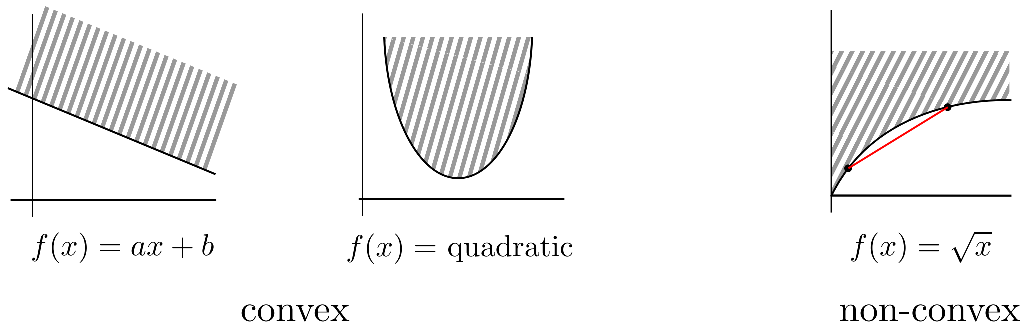

Figure 3: Examples of convex and non-convex functions.

C_2 (since each is convex), so this line segment also lies in their intersection. We conclude that every linear program has a convex feasible region.

For a relevant example that is more general than the finite intersection of half-spaces and subspaces, take C to be the set of $n \times n$ symmetric and positive semidefinite (PSD) matrices, viewed as a subset of $\mathbb{R}^{n^2}$.[7] It is clear that the set of symmetric matrices is convex — the average of symmetric matrices is again symmetric. It is true but less obvious that the set remains convex under the extra PSD constraint.[8] You'll work with symmetric PSD matrices in Part 3 of Homework #9.

4.2 Convex Functions

Who had the nerve to use the same word "convex" for two totally different things, sets and functions? The overloaded terminology becomes more forgivable if we define a function $f : \mathbb{R}^n \to \mathbb{R}$ to be *convex* if and only if the region above its graph is a convex set. See Figure 3 for some examples.

This definition is equivalent to the one given in Lecture #5, where we said that a convex function is one where all "chords" of its graph lie above the graph. Mathematically, this translates to

$$f(\lambda \mathbf{x} + (1-\lambda)\mathbf{y}) \leq \lambda f(\mathbf{x}) + (1-\lambda) f(\mathbf{y})$$

for every $\mathbf{x}, \mathbf{y} \in C$ and $\lambda \in [0,1]$. That is, for points $\mathbf{x}$ and $\mathbf{y}$, if you take the average of $\mathbf{x}$ and $\mathbf{y}$ and then apply f, you'll get a smaller number than if you first apply f to $\mathbf{x}$ and $\mathbf{y}$ and then average the results. It's not always easy to check whether or not a given function is convex, but there is a mature analytical toolbox for this purpose (taught in EE364, for example).

[7]There are many equivalent definitions of PSD matrices. One of the simplest is as the matrices of the form $\mathbf{A}^T\mathbf{A}$, like the covariance matrices we were looking at during our PCA discussions in Lectures #7–9.

[8]Another definition of PSD matrices is as the matrices $\mathbf{A}$ for which the corresponding quadratic form $\mathbf{x}^T\mathbf{A}\mathbf{x}$ is nonnegative for every $\mathbf{x} \in \mathbb{R}^n$. Using linearity, it is easy to see that the average of two matrices that satisfy this condition yields another matrix that satisfies the condition.

4.3 Convex Programs

Convexity leads to computational tractability. For example, in theory, it is possible to minimize an essentially arbitrary convex function over an essentially arbitrary convex feasible region. (There's a bit of fine print, but the conditions are quite mild.) This is fantastic news: in principle, we should be able to develop fast and robust algorithms for all of the convex optimization problems that we want to solve.

Practice is in the process of catching up with what the theory predicts. To oversimplify the current state-of-the-art, there are currently solvers that can handle medium-size and sufficiently nice convex optimization problems. The first piece of good news is that this is already enough to solve many problems that we're interested in; see Homework #9 for a couple of examples. The second piece of good news is that, as we speak, many smart people are working hard to close the gap in computational efficiency between linear and convex programming solvers — we expect major progress on convex solvers over the next 5 or so years.

Summarizing: convex programming is even more general than linear programming and captures some extra interesting applications. It is relatively computationally tractable, although the biggest instance sizes that can be solved are generally one or two orders of magnitude smaller than with linear programming (e.g., tens of thousands instead of millions).

Remark 4.1 (Why Convexity Helps) For intuition about why convexity leads to tractability, consider the case where the feasible region or the objective function is *not* convex. With a non-convex feasible region, there can be "locally optimal" feasible points that are not globally optimal, even with a linear objective function (Figure 4(left)). As we saw in Week 3, the same problem arises with a non-convex objective function, even when the feasible region is just the real line (Figure 4(right)). When both the objective function and feasible region are convex, this can't happen — all local optima are also global optima. This makes optimization much easier.

5 Application: Matrix Completion

5.1 Setup and Motivation

We conclude the lecture with a case study of convex programming, in the context of another central problem in sparse recovery with incomplete information: *matrix completion*. You saw this problem in Week #5, where we approached it using SVD-based techniques. Here, we'll obtain better results using convex optimization.

Recall the setup: there is an unknown "ground truth" matrix $\mathbf{M}$, analogous to the unknown sparse signal $\mathbf{z}$ in compressive sensing. The input is a matrix $\widehat{\mathbf{M}}$, derived from $\mathbf{M}$ by erasing some of its entries — the erased values are unknown, and the remaining values are known. The goal is to recover the matrix $\mathbf{M}$ from $\widehat{\mathbf{M}}$.

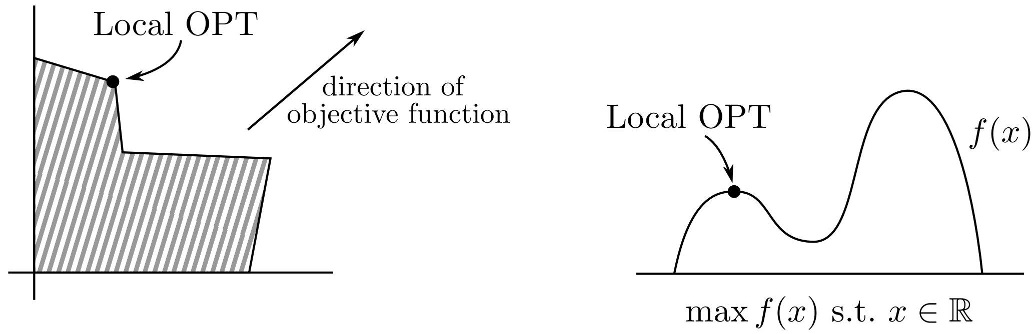

Figure 4: Non-convexity and local optima. (Left) A linear (i.e. convex) objective function with a non-convex feasible region. (Right) A non-convex objective function over a convex feasible region (the real line).

An example of matrix completion that received a lot of hype is the "Netflix challenge." Netflix was interested in the matrix $\mathbf{M}$ where rows are customers, columns are movies, and an entry of the matrix describes how much a customer would like a given movie. If a customer has rated a movie, then that entry is known; otherwise, it is unknown. Thus, most of the entries of $\mathbf{M}$ are missing in $\widehat{\mathbf{M}}$. Recovering $\mathbf{M}$ from $\widehat{\mathbf{M}}$, even approximately, would obviously be very useful to Netflix in designing a recommendation system.

Without any assumptions on the ground truth matrix $\mathbf{M}$, there is no way to recover its missing entries from $\widehat{\mathbf{M}}$ — they could be anything, and an algorithm would have no clue about what they are. A similar issue came up in compressive sensing, when we realized that there was no way to recover arbitrary unknown signals of length n while using fewer than n linear measurements. In compressive sensing, we made progress by assuming that the unknown signal was sparse. So what kind of assumption can play the role of sparsity in matrix completion? We could just assume that $\mathbf{M}$ is mostly zeroes, but then we get a stupid problem — presumably the best guess of $\mathbf{M}$ given $\widehat{\mathbf{M}}$ would just fill in all the missing entries with zeroes. This hack is unhelpful for the Netflix application, for example.

The key assumption we'll make is that the unknown matrix $\mathbf{M}$ has low rank. (One can also extend the following results to the case of matrices that are approximately low-rank.) For an extreme example, imagine that we knew that $\mathbf{M}$ was rank one, with all rows multiples of each other. In this case, as we saw in Lecture #9, we can sometimes recover $\mathbf{M}$ from $\widehat{\mathbf{M}}$ even when $\widehat{\mathbf{M}}$ has very few known entries.

5.2 Rank Minimization

Given that all we know about the unknown matrix $\mathbf{M}$ is that it agrees with $\widehat{\mathbf{M}}$ on the known entries and that it has low rank, we might try to recover $\mathbf{M}$ from $\widehat{\mathbf{M}}$ by solving the following optimization problem:

$$\min \operatorname{rank}(\mathbf{M}) \tag{14}$$

subject to

$$\mathbf{M} \text{ agrees with } \widehat{\mathbf{M}} \text{ on its known entries.} \tag{15}$$

This optimization problem has one real-valued decision variable for each unknown entry in $\widehat{\mathbf{M}}$; the known entries can be treated as constants.

Unfortunately, this rank-minimization problem is NP-hard, and no good general-purpose heuristic algorithms are known.[9] We confronted a similar issue in compressive sensing, where directly minimizing the sparsity of a solution to a linear system was an NP-hard problem. Our approach there was to relax the ℓ_0-minimization problem to the computationally tractable ℓ_1-minimization problem. Is there some way we can view matrix rank-minimization as an ℓ_0-minimization problem, and then switch to the ℓ_1 norm instead?

The singular value decomposition (SVD) provides an affirmative answer. Specifically, suppose the unknown $m \times n$ matrix $\mathbf{M}$ has the SVD

$$\mathbf{M} = \mathbf{U}\mathbf{S}\mathbf{V}^T,$$

where $\mathbf{U}$ is an $m \times m$ orthogonal matrix, $\mathbf{S}$ is an $m \times n$ diagonal matrix, and $\mathbf{V}$ is an $n \times n$ orthogonal matrix. Then, the rank of $\mathbf{M}$ is precisely the number r of non-zero singular values (i.e., entries of $\mathbf{S}$), with every row of $\mathbf{M}$ a linear combination of its top r right singular vectors, and every column of $\mathbf{M}$ a linear combination of its top r left singular vectors.

Writing $\Sigma(\mathbf{M})$ for the set of singular values of a matrix $\mathbf{M}$, we can therefore rephrase the optimization problem (14)–(15) as

$$\min |\text{supp}(\Sigma(\mathbf{M}))| \qquad (\text{a.k.a. } \|\Sigma(\mathbf{M})\|_0) \tag{16}$$

subject to

$$\mathbf{M} \text{ agrees with } \widehat{\mathbf{M}} \text{ on its known entries,} \tag{17}$$

which we can view as a ℓ_0-minimization problem.

5.3 Nuclear Norm Minimization

Following in our compressive sensing footsteps, we now consider the analogous ℓ_1-minimization problem, where we just change the 0-norm to the 1-norm:

$$\min \|\Sigma(\mathbf{M})\|_1 \tag{18}$$

subject to

$$\mathbf{M} \text{ agrees with } \widehat{\mathbf{M}} \text{ on its known entries.} \tag{19}$$

This optimization problem is called *nuclear norm minimization*.[10] It minimizes the sum of the singular values subject to consistency with the known information. Since ℓ_1-minimization

[9] Relatedly, its objective function is non-convex (i.e., "nasty"). For example, the average of rank-1 matrices need not be a rank-1 matrix (why?).

[10] One also hears about *trace minimization*, a closely related optimization problem.

promotes sparse solutions, we might hope that solving the problem (18)–(19) leads, under reasonable conditions, to the sparsest (i.e., minimum-rank) solution.

The following fact is non-obvious but can be proved using the convexity toolbox (e.g. from EE364) mentioned earlier.

Fact 5.1 *The objective function* (18) *is convex.*

Fact 5.1 implies that the optimization problem (18)–(19) is convex and hence can be solved relatively efficiently.

Since 2008 [2], there has been significant progress on identifying sufficient conditions on the matrix $\mathbf{M}$ and the number of known entries such that the optimization problem (18)–(19) successfully recovers $\mathbf{M}$. A typical guarantee is the following.

Theorem 5.2 ([9]) *Assume that:*

1. *The unknown matrix* $\mathbf{M}$ *has rank* r.
2. *The matrix* $\widehat{\mathbf{M}}$ *includes at least* $\Omega(r(m+n)\log^2(m+n))$ *known entries, chosen uniformly at random from* $\mathbf{M}$.
3. $\mathbf{M}$ *is sufficiently dense and non-pathological.*[11]

It is clear that some version of the second condition is needed — with too few known entries, there's no way to recover $\mathbf{M}$ from $\widehat{\mathbf{M}}$. For low-rank matrices $\mathbf{M}$, the required number of known entries is impressively small — sublinear in the number mn of $\mathbf{M}$'s entries. Given the small number of known entries, it is also clear that some version of the third condition is needed. If $\mathbf{M}$ is too sparse, like a diagonal matrix, then the randomly sampled known entries will likely all be zeroes.

References

[1] M. W. Browne. A Soviet discovery rocks world of mathematics. *New York Times*, November 7 1979.

[2] E. J. Candès and B. Recht. Exact matrix completion via convex optimization. *Foundations of Computational Mathematics*, 9(6):717–772, 2009.

[3] E. J. Candes, J. Romberg, and T. Tao. Robust uncertainty principles: exact signal reconstruction from highly incomplete frequency information. *IEEE Transactions on Information Theory*, 52(2):489–509, 2006.

[4] E. J. Candes, J. Romberg, and T. Tao. Stable signal recovery from incomplete and inaccurate measurements. *Communications on Pure and Applied Mathematics*, 59(8):1207–1223, 2006.

[11] The formal term is "incoherent," which roughly means that the rows of $\mathbf{M}$ are not well-aligned with the standard basis vectors. This is similar to the assumption on the measurement matrix in Theorem 1.1.

[5] G. W. Dantzig. *Linear Programming and Extensions*. Princeton, 1963.

[6] D. L. Donoho. Compressed sensing. *IEEE Transactions on Information Theory*, 52(4):1289–1306, 2006.

[7] L. G. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20(1):191–194, 1979.

[8] J. Matousek and B. Gärtner. *Understanding and Using Linear Programming*. Springer, 2006.

[9] B. Recht. A simpler approach to matrix completion. *Journal of Machine Learning Research*, 12:3413–3430, 2011.