Chapter 3

Sparse solutions of linear equations (Project A)

3.1 Introduction

Suppose we have m linear equations in n unknowns $x_1, x_2, \ldots, x_n$:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \ldots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \ldots & + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & \vdots & & \vdots & & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \ldots & + & a_{mn}x_n & = & b_m. \end{array}$$

Writing $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \ldots & x_n \end{bmatrix}^T$ and $\mathbf{b} = \begin{bmatrix} b_1 & b_2 & \ldots & b_m \end{bmatrix}^T$, we can write the system of equations as the matrix equation

$$\mathbf{Ax} = \mathbf{b}, \tag{3.1}$$

where the entries of the $m \times n$ matrix $\mathbf{A}$ are the $\{a_{ij}\}$. In what follows, we assume that $\mathbf{x}$, $\mathbf{b}$ and $\mathbf{A}$ are all real-valued.

In this project we are interested in the *underdetermined* case in which $m < n$. As we all know, such a system of equations may have infinitely many solutions. Suppose, however, that we have additional information: namely that there is a solution $\mathbf{x}^*$ to the system of equations (3.1) which is *sparse*, meaning that only a few of its entries are nonzero. To be more precise, let's suppose we know that it has k nonzeros. We will say that a vector with k nonzero coefficients has *sparsity* k. To clarify, the solution $\mathbf{x}^*$ is unknown to us and we do not know which of its k coefficients are nonzero: we just know that such a solution with sparsity k exists. This project explores some ways we might go about finding it. In particular, we will explore what happens when the entries of the matrix $\mathbf{A}$ are i.i.d. instances of standard Normal random variables. In this case, it turns out that there are infinitely many solutions to (3.1) with probability 1 when $m < n$.

3.2 A very simple algorithm

Let's think about what we know from linear algebra. If the matrix $\mathbf{A}$ is full rank, so that $\text{rank}(\mathbf{A}) = m$, then we can give a closed form expression for one solution to (3.1). It is easy to check that $\bar{\mathbf{x}} = \mathbf{A}^\dagger \mathbf{b}$ is a solution, where $\mathbf{A}^\dagger := \mathbf{A}^T(\mathbf{A}\mathbf{A}^T)^{-1}$ denotes the *Moore-Penrose pseudoinverse* of $\mathbf{A}$, which can be calculated efficiently in MATLAB using the command `pinv(A)`. However, if there are infinitely many solutions, $\bar{\mathbf{x}}$ is unlikely to be equal to $\mathbf{x}^*$, and unlikely to even be sparse.

What we could try is to find the 'closest' vector to $\bar{\mathbf{x}}$ which has at most k nonzero coefficients. We can think of 'closeness' in terms of the Euclidean distance, in which case we are wanting to find some vector $\mathbf{z}$ which minimizes

$$\|\mathbf{z} - \mathbf{x}\|_2^2 = \sum_{i=1}^{n} (z_i - \bar{x}_i)^2.$$

It is easy to show (check you agree!) that the solution $\mathbf{z}$ is given by keeping the k entries of $\bar{\mathbf{x}}$ with largest absolute value and setting the rest of the entries to zero. Let's write $\mathcal{S}_k : \mathbb{R}^n \to \mathbb{R}^n$ for the *sparsifying* function which performs this operation. We can think of this operation as a 'projection' onto the set of vectors with sparsity at most k.

Note that if some of the coefficients are equal, there could be ambiguity about the ordering of the magnitude of the entries. For the purposes of this project, we will use the `sort` function to obtain an ordering, which will have its own tie-breaking rule. In practice, the issue of equal coefficients will not occur in this project since the coefficients will be generated from continuous random variables.

The following is code for a MATLAB function `sparsify.m` which takes two arguments, a (column) vector `x` and a sparsity `k` and output a single argument which is the (column) vector `x` with all but the largest `k` coefficients in magnitude set to zero.

```
function x = sparsify(x,k)
    [xsort ind] = sort(abs(x),'descend');
    x(ind(k+1:end),1) = 0;
end
```

Exercise 3.1. *Use the `randn` function to generate a random Normally distributed vector `x` of length 20 and run the `sparsify.m` function on `x` with `k=7`. Display the input and the output vectors.*

Let's precisely define the problem we will explore.

Problem 1.

- *We generate $\mathbf{x}^*$, a vector of length n which has k nonzero entries in (uniformly) random positions, and where each of these coefficients are i.i.d. uniformly distributed on $[0, 1]$ (with the rest of the entries being zero). You may find the `randsample` and `rand` functions useful in generating such a vector.*
- *We then generate an $m \times n$ matrix $\mathbf{A}$ whose entries are i.i.d. standard Normal and calculate $\mathbf{b} = \mathbf{A}\mathbf{x}^*$. Such a matrix can be generated in MATLAB using the command `A=randn(m,n)`.*
- *We then see how well we can recover $\mathbf{x}^*$ from $\mathbf{A}$ and $\mathbf{b}$, assuming that we also know the sparsity k.*

Our first simple algorithm is the one we just discussed, and it is summarized in Algorithm 1.

We need a way of assessing the accuracy of the solution. Let's take the Euclidean norm of the difference between the obtained solution $\tilde{\mathbf{x}}$ and the actual solution $\mathbf{x}^*$ used to generate the equations, and let's normalize it by dividing by the Euclidean norm of $\mathbf{x}^*$. In other words, our error metric is

$$\mathcal{E} := \frac{\|\tilde{\mathbf{x}} - \mathbf{x}^*\|_2}{\|\mathbf{x}^*\|_2}.$$

Algorithm 1 One Step Sparsification

```
Inputs: $\mathbf{A}$, $\mathbf{b}$ and $k$.
Outputs: A $k$-sparse vector $\tilde{\mathbf{x}}$.
  1. Set $\bar{\mathbf{x}} := \mathbf{A}^\dagger \mathbf{b}$.
  2. Set $\tilde{\mathbf{x}} := \mathcal{S}_k(\bar{\mathbf{x}})$.
```

Exercise 3.2. *Write a* MATLAB *function* `one_step.m` *which takes three inputs k, m and n, generates an instance of Problem 1, runs Algorithm 1, and outputs a single argument giving the error $\mathcal{E}$.*

Exercise 3.3. *Test your function with input values $k = 30$, $m = 200$ and $n = 400$. Display the error $\mathcal{E}$ (but do not display long vectors or matrices in your output!).*

3.3 An iterative algorithm

You should have observed that the One Step Sparsification algorithm is unable to give a very good approximation to $\mathbf{x}^*$. In this section, we explore an iterative approach which builds upon the One Step Sparsification algorithm.

We observed before that $\mathcal{S}_k$ can be thought of as a 'projection' onto the set of vectors with sparsity at most k. Now we also consider projecting onto the set of vectors for which the equations hold. This set $\{\mathbf{x} \in \mathbb{R}^n : \mathbf{A}\mathbf{x} = \mathbf{b}\}$ is in fact a subspace of $\mathbb{R}^n$. Given a vector $\mathbf{z}$, its Euclidean projection onto $\{\mathbf{x} \in \mathbb{R}^n : \mathbf{A}\mathbf{x} = \mathbf{b}\}$ is the solution to the optimization problem

$$\bar{\mathbf{z}} := \operatorname*{argmin}_{\mathbf{x} \in \mathbb{R}^n} \|\mathbf{x} - \mathbf{z}\|_2 \quad \text{subject to} \quad \mathbf{A}\mathbf{x} = \mathbf{b}.$$

An interesting exercise[1] is to show that this optimization problem has the closed-form solution

$$\bar{\mathbf{z}} := \mathbf{z} + \mathbf{A}^\dagger(\mathbf{b} - \mathbf{A}\mathbf{z}).$$

We now have the tools we need for an iterative algorithm. We are seeking to find a vector which both satisfies the equations $\mathbf{A}\mathbf{x} = \mathbf{b}$ and which has sparsity at most k. Or to put it another way, we want to find a vector which is in the intersection of two sets: $\{\mathbf{x} \in \mathbb{R}^n : \mathbf{A}\mathbf{x} = \mathbf{b}\}$ and $\{\mathbf{x} \in \mathbb{R}^n : \mathbf{x} \text{ has sparsity at most } k\}$. One way to do this is to alternately project onto each of the sets and keep iterating – this is known as the Alternating Projection algorithm, given in Algorithm 2. The algorithm generates a sequence of iterates $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$. An initial $\mathbf{x}_0$ must be chosen: we will simply take $\mathbf{x}_0 = \mathbf{0}$, the zero vector. Note that with this choice the first iteration of Alternating Projection is nothing other than the One Step Sparsification algorithm. We must also decide how many iterations to run the algorithm for, so we must choose a parameter I for the number of iterations.

Exercise 3.4. *Write a* MATLAB *function* `alt_proj.m` *which takes three inputs k, m and n, generates an instance of Problem 1, runs Algorithm 2, and outputs a single argument giving the error $\mathcal{E}$. Take the number of iterations to be $I = 500$.*

Exercise 3.5. *Test your function with input values $k = 30$, $m = 200$ and $n = 400$. Display the error $\mathcal{E}$ (but do not display long vectors or matrices in your output).*

[1]You can use the method of Lagrange multipliers for example.

```
Algorithm 2 Alternating Projection
Inputs: $\mathbf{A}$, $\mathbf{b}$ and $k$.
Outputs: A $k$-sparse vector $\tilde{\mathbf{x}}$.
Initialize: $\mathbf{x}_0 = \mathbf{0}$ and $i = 0$.
While $i < I$:
    1. Set $\bar{\mathbf{x}} := \mathbf{x}_i + \mathbf{A}^\dagger(\mathbf{b} - \mathbf{A}\mathbf{x}_i)$.
    2. Set $\mathbf{x}_{i+1} := \mathcal{S}_k(\bar{\mathbf{x}})$.
    3. Set $i := i + 1$.
Set $\tilde{\mathbf{x}} := \mathbf{x}_i$.
```

3.4 Systematic testing

You hopefully found that the Alternating Projection algorithm finds a much more accurate approximation to $\mathbf{x}^*$. In fact, you should find that it is possible to approximate $\mathbf{x}^*$ to any level accuracy by increasing the number of iterations I sufficiently. What we are observing then is a phenomenon of *exact recovery.* In this section, we want to explore for what parameters k, m and n this phenomenon is observed. To do this, let's fix $m = 200$ and $n = 400$ as before, but vary the sparsity parameter k. We will run the algorithm for $I = 500$ iterations and declare that we have successful recovery if $\mathcal{E} < \epsilon$ for some $\epsilon > 0$. We want to estimate the probability that recovery is successful, so to do that we will need to run multiple trials and record the proportion of successes.

Exercise 3.6. *Take $I = 500$ and $\epsilon = 10^{-4}$. Alter your Alternating Projection algorithm so that it instead outputs a logical variable `success` which takes the value 1 if $\mathcal{E} < \epsilon$ and 0 otherwise. Call your new function `alt_proj2.m`. Test your function with input values $k = 30$, $m = 200$ and $n = 400$. Display the output.* Hint: the pseudoinverse can be calculated just once outside of the iteration loop.

Exercise 3.7. *Take $m = 200$ and $n = 400$ as before. Set $I = 500$ and $\epsilon = 10^{-4}$ as before. Vary k in steps of 10 between 10 and 200 inclusive. For each k, run the Alternating Projection algorithm on 100 different random test problems by running your `alt_proj2.m` function 100 times[2]. Plot a graph of proportion of successes against k. Explain what the plot shows about the exact recovery phenomenon for $m = 200$ and $n = 400$.*

3.5 Exploiting positivity

The nonzero coefficients of our vector $\mathbf{x}^*$ have been drawn from the uniform distribution on $[0, 1]$, which actually gives us even more information. Not only do we know that $\mathbf{x}^*$ has sparsity k, but we also know that its k nonzero coefficients are all in the range $[0, 1]$. Your final challenge is to alter the Alternating Projection algorithm to exploit this information so that it performs better. This can be done by simply altering the sparsification step so that it find the closest vector which has sparsity at most k *and* whose coefficients are all in the range $[0, 1]$.

Exercise 3.8. *Alter the `sparsify.m` code so that it instead finds the closest vector with at most k nonzero coefficients which are all in the range $[0, 1]$. Call your new function*

[2]Note that your code may take some time to run.

`sparsify2.m`. Repeat Exercise 3.1 using your new function (it does not need to be the same random vector).

Exercise 3.9. *Alter `alt_proj2.m` by replacing `sparsify` with `sparsify2`. Give your altered code a new name (anything you like). Repeat the experiment in Exercise 3.7 using your new code. Explain whether you observe an improvement in recovery performance.*