Lecture 21
Beyond Least-Square Fitting: The L^1 Norm

Jason J. Bramburger

As we have alluded to in class, there are a lot of ways to measure how 'big' a vector is. We typically use the 2-norm to calculate distance in Euclidean space, and distance from the origin essentially measures the size of the vector. For example, let's say we have two airplanes. One of them has traveled 25 kilometres north, 30 kilometres east, and 1 kilometres up. The other has traveled 35 kilometres north, 10 kilometres east, and 2 kilometres up. We can encode this information using the vectors

$$a_1 = \begin{bmatrix} 25 & 30 & 1 \end{bmatrix}, \qquad a_2 = \begin{bmatrix} 35 & 10 & 2 \end{bmatrix},$$

so that the vectors point to the current position of the airplanes. Assuming we are at the origin, which is farther from us? We can use the 2-norm to determine this:

$$\|a_1\|_2 = \sqrt{25^5 + 30^2 + 1^2} \approx 39,$$
$$\|a_2\|_2 = \sqrt{35^5 + 10^2 + 2^2} \approx 36.$$

So, the first airplane is farther from us.

Here's another unrelated example. Suppose we collect the final grades from two different classes with 5 people in them. The grades are given by

$$g_1 = \begin{bmatrix} 82 & 86 & 90 & 71 & 74 \end{bmatrix}, \qquad g_2 = \begin{bmatrix} 91 & 76 & 63 & 85 & 81 \end{bmatrix}.$$

We could ask ourselves which class did better overall. One way to do this would be to calculate the class averages (add the grades up and divide by 5). Now, since they are both being divided by five, we can ignore this part of the routine and simply just see which has the largest sum. This leads to the idea of a 1-norm:

$$\|g_1\|_1 = |82| + |86| + |90| + |71| + |74| = 403,$$
$$\|g_2\|_1 = |91| + |76| + |63| + |85| + |81| = 396.$$

I'm including absolute values here because the 1-norm has to be able to handle negative numbers in vectors as well. Essentially, the 1-norm measures the distance of each component of a vector from 0 and sums up these distances.

Okay, one more example. Suppose you are competing against a friend in a javelin throw competition. You each get three throws and at the end the winner is whoever threw the javelin the furtherest over all throws. In metres the distances thrown are:

$$d_1 = \begin{bmatrix} 82 & 73 & 79 \end{bmatrix}, \qquad d_2 = \begin{bmatrix} 81 & 78 & 83 \end{bmatrix}.$$

So, who won? Well, the winner is the person with the longest throw: person 2. What you're doing in your head is calculating the infinity (or max) norm:

$$\|g_1\|_\infty = \max\{|82|, |73|, |79|\} = 82,$$
$$\|g_2\|_\infty = \max\{|81|, |78|, |83|\} = 83.$$

Again, the absolute value is in here to handle negative values. Essentially, the infinity norm asks for the value in the vector that is the farthest from 0. Compare this with the 1-norm where to be big you (roughly) want to have all of components far from 0, while the infinity norm just asks for one of them to be far from 0 and the rest don't matter.

These examples illustrate some basic facts to you. When we ask how big a vector is, there are many different ways to answer this question and it is context dependent. I could list infinitely many more measures for vectors,

but the three presented above will be the main ones you encounter in and out of this class. You should also recall that we already encountered some different norms when we talked about the SVD, but those were for matrices. Those norms ask and answer similar questions to those above, just about different objects (matrices).

Curve Fitting

Let's start by discussing **linear regression** or, as it's typically known outside of math, the **line of best fit**. Suppose we have some data points

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n).$$

Our goal is to find a linear function

$$y(x) = mx + b$$

that minimizes the error (distances) between the data points and the line. We can define the error vector as

$$\begin{bmatrix} y(x_1) - y_1 & y(x_2) - y_2 & \ldots & y(x_n) - y_n \end{bmatrix}.$$

So, the question is, how do we measure how large the error is? In general, for any $p \geq 1$ we can define an error function using the p-norm

$$E_p = \left(\sum_{j=1}^{n} |y(x_j) - y_j|^p \right)^{1/p}.$$

Notice than when $p = 1$ we get the 1-norm presented above and with $p = 2$ we get the usual Euclidean distance. You can choose any value of $p \geq 1$ and get a slightly different measure of the error. The standard method of finding the line of best fit is to minimize E_2, that is the 2-norm of the error vector. This is referred to as a **least squares** fit. We are going to compare the cases of $p = 1$ and $p = 2$. Let's start with some data.

```
x=[0.1 0.4 0.7 1.2 1.3 1.7 2.2 2.8 3.0 4.0 4.3 4.4 4.9];
y=[0.5 0.9 1.1 1.5 1.5 2.0 2.2 2.8 2.7 3.0 3.5 3.7 3.9];
```

We can define error functions for various p and then use finminsearch to find the coefficients m and b that minimize the error. I'll include the $p = 10$ error norm just for fun here.

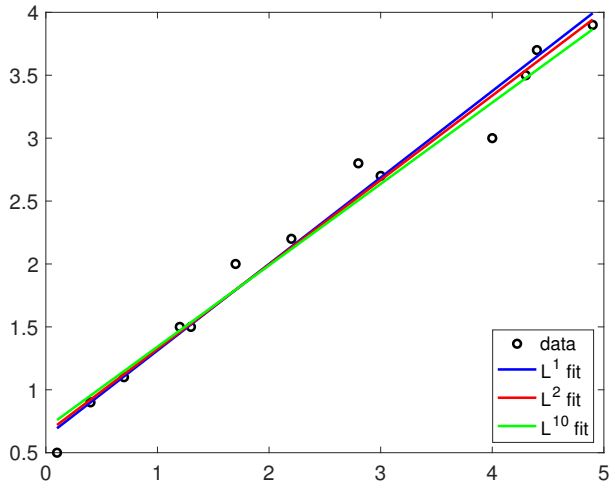
```
line = @ (m,b) m*x+b;
E1 = @ (par) norm(y - line(par(1),par(2)),1); % par = parameter = (m,b)
E2 = @ (par) norm(y - line(par(1),par(2)),2);
E10 = @ (par) norm(y - line(par(1),par(2)),10); % just for fun
coeff_L1 = fminsearch(E1, [1 1]); % Initial guess for par = [1 1]
coeff_L2 = fminsearch(E2, [1 1]);
coeff_L10 = fminsearch(E10, [1 1]);
```

The results are as follows:

$$\begin{aligned} \text{coeff\_L1} &= \begin{bmatrix} 0.6873 & 0.6251 \end{bmatrix} \\ \text{coeff\_L2} &= \begin{bmatrix} 0.6713 & 0.6530 \end{bmatrix} \\ \text{coeff\_L10} &= \begin{bmatrix} 0.6470 & 0.6950 \end{bmatrix} \end{aligned}$$

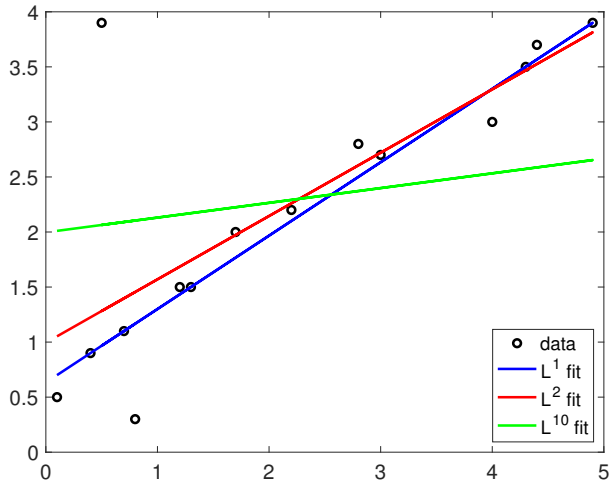
Let's plot the lines along with the data to get a comparison.

```
figure(1)
plot(x,y,'ok','Linewidth',2)
hold on
plot(x,line(coeff_L1(1),coeff_L1(2)),'b','Linewidth',2)
plot(x,line(coeff_L2(1),coeff_L2(2)),'r','Linewidth',2)
plot(x,line(coeff_L10(1),coeff_L10(2)),'g','Linewidth',2)
legend('data','L^1 fit','L^2 fit', 'L^{10} fit', 'Location', 'southeast')
set(gca,'Fontsize',16)
```



It's difficult to say which is best since they look so similar. However, they are quite different in how they handle outliers. Let's add two massive outliers to the data and see how things change.

```
x = [x 0.5 0.8];
y = [y 3.9 0.3];

line = @ (m,b) m*x+b;
E1 = @ (par) norm(y - line(par(1),par(2)),1); % par = parameter = (m,b)
E2 = @ (par) norm(y - line(par(1),par(2)),2);
E10 = @ (par) norm(y - line(par(1),par(2)),10); % just for fun
coeff_L1 = fminsearch(E1, [1 1]); % Initial guess for par = [1 1]
coeff_L2 = fminsearch(E2, [1 1]);
coeff_L10 = fminsearch(E10, [1 1]);

figure(2)
plot(x,y,'ok','Linewidth',2)
hold on
plot(x,line(coeff_L1(1),coeff_L1(2)),'b','Linewidth',2)
plot(x,line(coeff_L2(1),coeff_L2(2)),'r','Linewidth',2)
plot(x,line(coeff_L10(1),coeff_L10(2)),'g','Linewidth',2)
legend('data','L^1 fit','L^2 fit', 'L^{10} fit', 'Location', 'southeast')
set(gca,'Fontsize',16)
```



The resulting coefficients are as follows:

$$\begin{aligned} \text{coeff\_L1} &= \begin{bmatrix} 0.6668 & 0.6333 \end{bmatrix} \\ \text{coeff\_L2} &= \begin{bmatrix} 0.5752 & 0.9947 \end{bmatrix} \\ \text{coeff\_L10} &= \begin{bmatrix} 0.1340 & 1.9968 \end{bmatrix} \end{aligned}$$

Comparing the coefficients and the graphs with the data without the outliers shows that the L^1 norm didn't change much. In fact, it follows the original data pretty well. For larger values of p you are raising the error $|y(x_j) - y_j|$ to a larger power, so big values get bigger. Hence, the bigger p is, the more emphasis we are putting on outliers. The difference between the 1- and 2-norm is going to have big implications for the rest of the chapter. We are going to explore more examples where the L^1 norm will offer an advantage.

Linear Systems

We are now going to focus on systems of linear equations, written in matrix form as

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

Recall that such linear systems can have exactly one solution, no solution, or infinitely many solutions. Moreover, when $\mathbf{A}$ is an invertible square matrix you can easily obtain the unique solution by inverting $\mathbf{A}$ to get

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}.$$

The question is, what about when there isn't a unique solution?

Underdetermined Systems

We will start with the case of infinitely many solutions. This is most common when there are more variables than equations (more columns in $\mathbf{A}$ than rows). For example, we can consider the equation

$$0.5x_1 + 0.25x_2 + 3x_3 + 7x_4 - 2x_5 + 10x_6 = 3.$$

We only have one equation, but 6 unknowns! Therefore, the best we could do is write one of the variables as a function of the other five. In matrix form this is represented by the matrix and vector

$$\begin{aligned} \mathbf{A} &= \begin{bmatrix} 0.5 & 0.25 & 3 & 7 & -2 & 10 \end{bmatrix} \\ \mathbf{b} &= \begin{bmatrix} 3 \end{bmatrix} \end{aligned}$$

So, if there are infinitely many solutions, how can we solve the problem? To start, let's pretend we don't know there are infinitely many solutions and give it to MATLAB to solve. We can use the backslash command.

```
clear all; close all; clc

A = [1/2 1/4 3 7 -2 10];
b = [3];
x = A\b;
```

MATLAB gives the solution

$$\mathbf{x} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0.3 \end{bmatrix}.$$

We won't talk about what the backslash command does to solve this problem yet. For now, you should notice that the solution is **sparse** - it has a lot of zeros.

Another way to solve this system would be to do what we do with square matrices: try to invert it. The problem is, if you use the inv() command in MATLAB, you'll get an error. Rightfully so. What is the inverse of a non-square matrix? There is actually an analogue of inverses for non-square matrices called the *pseudoinverse*. It comes from our old friend the SVD! If you take the SVD of $\mathbf{A}$ you get

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^*.$$

Remember that $\mathbf{U}$ and $\mathbf{V}$ are orthogonal/unitary, so they are invertible. What do we do with Σ then? Well, it is a diagonal matrix, so we could invert it the way we usually invert diagonal matrices: take the σ_j on the diagonal and replace them with $1/\sigma_j$ if they are positive. Otherwise, we just leave the zero diagonal elements of Σ as zeros for the inverse. This leads to the **Moore–Penrose** inverse of Σ, denoted $\Sigma^\dagger$. Then, using the inverses of $\mathbf{U}$ and $\mathbf{V}^*$, we obtain the Moore–Penrose inverse (or pseudoinverse) of $\mathbf{A}$, again denoted $\mathbf{A}^\dagger$:

$$\mathbf{A}^\dagger = \mathbf{V}\Sigma^\dagger\mathbf{U}^*.$$

In MATLAB we can execute this with the pinv() command.

```
x2 = pinv(A)*b;
```

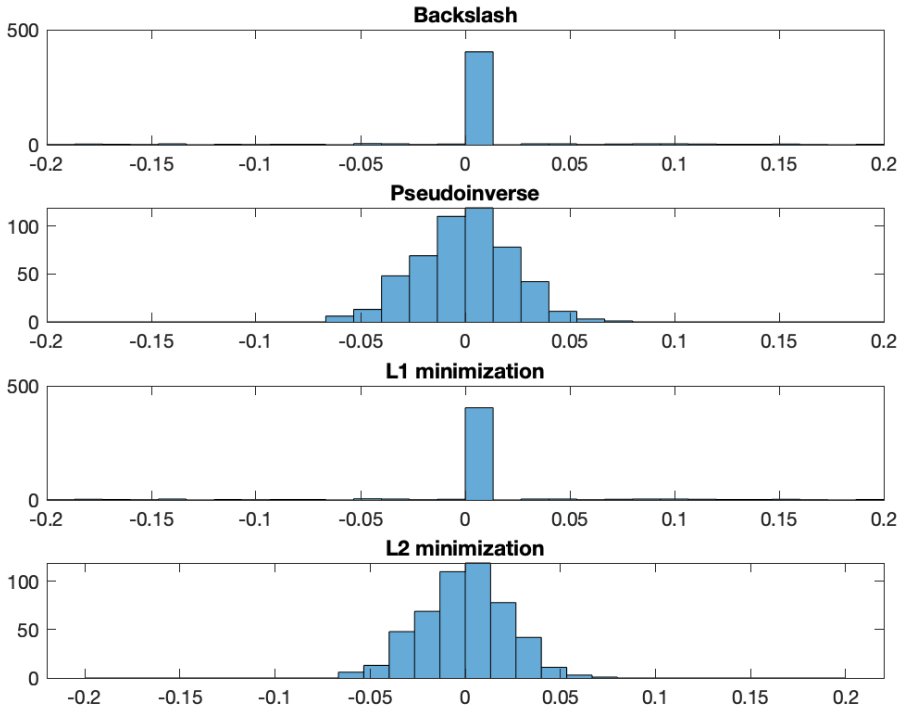
Now MATLAB returns the solution to the underdetermined linear system

$$\mathbf{x} = \begin{bmatrix} 0.0092 \\ 0.0046 \\ 0.0554 \\ 0.1294 \\ -0.0370 \\ 0.1848 \end{bmatrix}.$$

This is again a valid solution to the system, but it isn't sparse. So, how can you decide which of the infinitely many solutions you want returned? One criteria might be that you want the "smallest" solution, i.e. the one that minimizes some norm. We will experiment with a much larger system (100 equations and 500 variables). We will also compute solutions with the backslash and pseudoinverse commands and compare all four solutions by looking at histograms. In order to minimize the L^1 and L^2 norm, we will use a convex optimization package that can be downloaded online at: **http://cvxr.com/cvx/**. After downloading, open the folder and type **cvx_setup** into the command window. This will get you started with cvx. We can give it what we want to minimize as well as any constraints. The top line specifies the name of the optimization variable and its dimension.

```
% Initializing a 100x500 linear system
m=100; n=500;
A=randn(m,n);
b=randn(m,1);

% Built-in MATLAB solvers
x1=A\b;
x2=pinv(A)*b;

% cvx solutions
cvx_begin quiet
    variable x3(n);
    minimize( norm(x3,1) );
    subject to
    A*x3 == b;
cvx_end
cvx_begin quiet
    variable x4(n);
    minimize( norm(x4,2) );
    subject to
    A*x4 == b;
```

```
cvx_end

% plot histograms
figure(3)
subplot(4,1,1)
histogram(x1,linspace(-.2,.2,31)) % specify edges of bins
title('Backslash')
xlim([-.2 .2])
subplot(4,1,2)
histogram(x2,linspace(-.2,.2,31))
title('Pseudoinverse')
xlim([-.2 .2])
subplot(4,1,3)
histogram(x1,linspace(-.2,.2,31))
title('L1 minimization')
xlim([-.2 .2])
subplot(4,1,4)
histogram(x4,linspace(-.2,.2,31))
title('L2 minimization')
```



Let's talk about the four methods:

1. The solutions using pinv() and minimizing the 2-norm are the same! This is exactly what pinv() does.
2. The 1-norm minimization solution is sparse, as is the backslash.
3. The way backslash chooses a solution is that it finds one with at least $n - m$ zeros (400 in this case).
4. The 1-norm minimization and backslash are both sparse, but they are not the same. The 1-norm minimization gives many values close to zero but not exactly zero.

```
nnz(x1)
nnz(x3)
```

Overdetermined Systems

Okay, so how about overdetermined systems? That is, systems with no solution. This normally occurs when there are more equations than variables (more rows than columns). First, if there is no solution, what does it mean to "solve" such a system. We will try and get as close as possible to a solution by minimizing the residual:

$$\mathbf{r} = \mathbf{A}\mathbf{x} - \mathbf{b}.$$

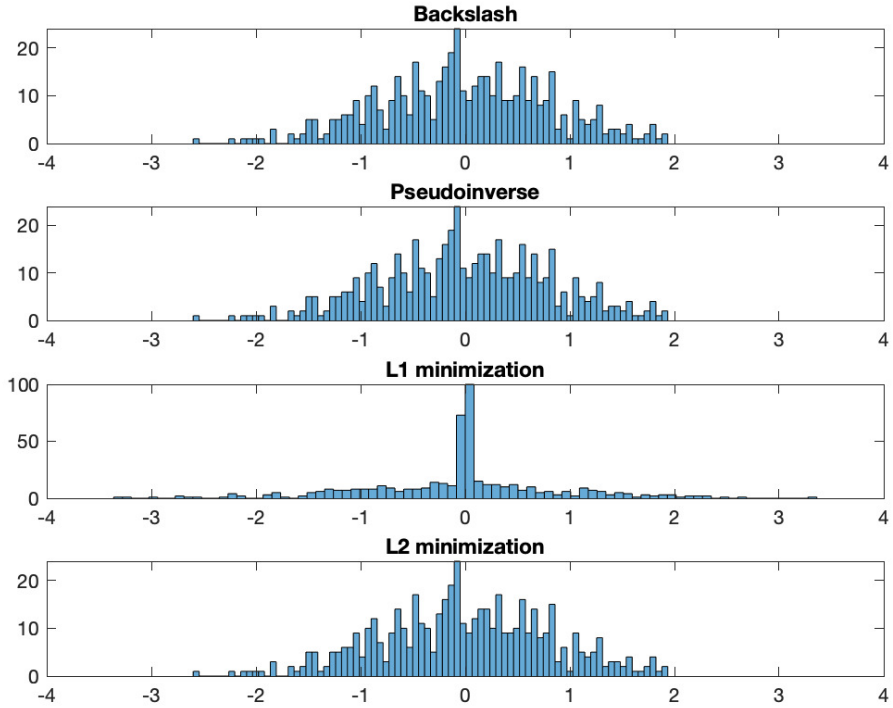
This puts us back in a similar situation to before. We need to specify what norm we will minimize the residual with respect to. Note that we have already solved an overdetermined system in this lecture. The line of best fit was overdetermined! We were looking for a vector $\mathbf{x}$ with the parameters m and b that go through every point:

$$\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{bmatrix} \cdot \begin{bmatrix} b \\ m \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

The residual vector is precisely the error vector that we wrote down earlier. Let's solve this linear system using backslash, pinv(), L^1 minimization, and L^2 minimization. We will have 500 equations with 150 variables.

```
clear all; close all; clc

m=500; n=150;
A=randn(m,n);
b=randn(m,1);

% Built-in MATLAB solvers
x1=A\b;
x2=pinv(A)*b;

% cvx solutions
cvx_begin quiet
    variable x3(n);
    minimize( norm(A*x3-b,1) );
cvx_end
cvx_begin quiet
    variable x4(n);
    minimize( norm(A*x4-b,2) );
cvx_end

% Plot historgrams
figure(4)
subplot(4,1,1)
histogram(A*x1-b,80) % specify number of bins
title('Backslash')
xlim([-4 4])
subplot(4,1,2)
histogram(A*x2-b,80)
title('Pseudoinverse')
xlim([-4 4])
subplot(4,1,3)
histogram(A*x3-b,80)
title('L1 minimization')
xlim([-4 4])
subplot(4,1,4)
histogram(A*x4-b,80)
title('L2 minimization')
xlim([-4 4])
```

Notice that now only the L^1 minimization is sparse. All of the others are exactly the same: they all minimize the L^2 residual.



In Closing: The main idea behind compressed sensing is that we are going to try to reproduce some signal or image with as little information as possible. In order to do so, we have to take the perspective that something about the signal is sparse. We will use the L^1 norm to promote sparsity.