10-725: Optimization **Fall 2012**

Lecture 2: August 30

Lecturer: Geoff Gordon/Ryan Tibshirani *Scribes: Ahmed Hefny,Dougal Sutherland, Guanyu Wang*

2.1 Sparse Linear Regression

We first provide an example problem that shows some advantages of casting a problem in the form of convex optimization.

Suppose we have a response $y \in \mathbb{R}^n$ (i.e. a scalar response for each of n training examples) and predictors $A = (A_1, \dots, A_p) \in \mathbb{R}^{n \times p}$ (i.e. p-dimensional features for each of n training examples). We wish to use a linear model $y \approx Ax$, where $x \in \mathbb{R}^p$.

If $n > p$, this is the classical linear regression problem

$$\min_{x \in \mathbb{R}^n} \|y - Ax\|^2. \tag{2.1}$$

The solution to this problem is well-defined and easy to find if $n > p$. But in some applications $n \ll p$, and we further believe that many of the extracted features $A_1, \dots, A_p$ could be irrelevant. We therefore wish to find a model x with many zero coefficients, as illustrated in Figure 2.1.

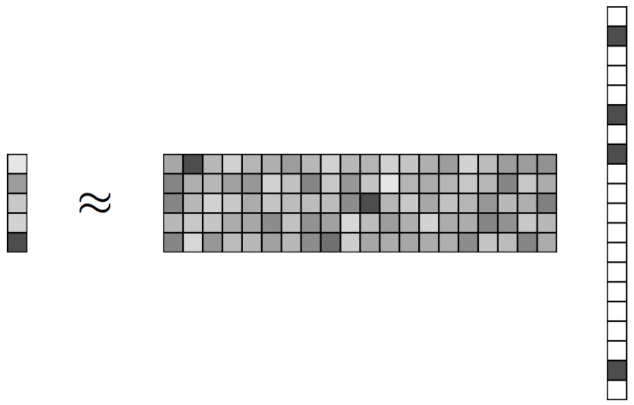

Figure 2.1: The linear regression model $y \approx Ax$, with $n \ll p$ and a sparse x.

For example, if y is the size of a tumor, it might be reasonable to suppose that it can be expressed as a linear combination of genetic information in A, but we expect that most genes will be unimportant to this combination: therefore most components of x will be zero.

2.1.1 Best Subset Selection

One natural idea for incorporating sparsity into (2.1) is to constrain the number of nonzero components of x to be no more than some k:

$$\min_{x \in \mathbb{R}^n} \|y - Ax\|^2 \text{ subject to } \|x\|_0 \leq k, \tag{2.2}$$

where $\|x\|_0$, the 0-norm, denotes the number of nonzero components in x.

This "norm" is non-convex, however, which makes solving (2.2) difficult (in fact, NP-hard). Figure 2.2 shows a sublevel set in $\mathbb{R}^2$, illustrating the nonconvexity. The only known algorithm is essentially brute-force evaluation of all possible subsets; the solution cannot be computed in practice for $p \gtrsim 40$. We also know very little about statistical properties of the solution.

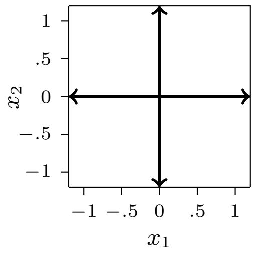

Figure 2.2: Sublevel set for the constraint of (2.2): $\{x \in \mathbb{R}^2 : \|x\|_0 \leq 1\}$.

We call this the 0-norm because it is a limit case of the general p-norm. A vector $x = (x_1, x_2, \cdots, x_n)$ has p-norm $\|x\|_p = \left(\sum_{i=1}^n |x_i|^p\right)^{1/p}$. Figure 2.3 shows contours of different p-norms in $\mathbb{R}^2$. Note that the non-convex contour in Figure 2.3 is just a sketch for p-norm with $p < 1$, and that in fact the p-norm is not a norm for $p < 1$.

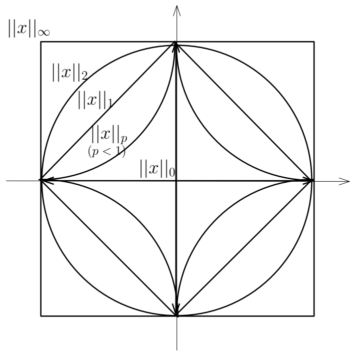

Figure 2.3: An illustration for different p-norms in 2D.

2.1.2 Forward Stepwise Regression

Another natural approach is given by the following algorithm:

1. Find the variable j such that $|A_j^T y|$ is largest. If the variables have been centered to have zero mean and scaled to have unit variance, then $A_j^T y$ is the correlation between predictors A_j and response y.

2. Update x_j by regressing y onto A_j, so that we are solving $\min_{x_j \in \mathbb{R}} \|y - A_j x_j\|^2$.

3. Find a new variable $k \neq j$ such that $|A_k^T r|$ is largest, where r is the residual $y - A_j x_j$. This finds the variable most correlated to the residuals.

4. Update x_j, x_k by regressing y onto A_j, A_k.

5. Repeat until the desired number of variables have been selected.

It is unknown whether this algorithm minimizes some criterion function. A few statistical properties of this estimate are known, such as its consistency and some details about the convergence rate, but the proofs are relatively complicated and many properties remain unknown.

2.1.3 Lasso

Instead of the 0-norm of best subset selection, we can also use the (convex) 1-norm $\|x\|_1 = \sum_{i=1}^p |x_i|$:

$$\min_{x \in \mathbb{R}^n} \|y - Ax\|^2 \text{ subject to } \|x\|_1 \leq t. \tag{2.3}$$

A sublevel set in $\mathbb{R}^2$ is shown in Figure 2.4.

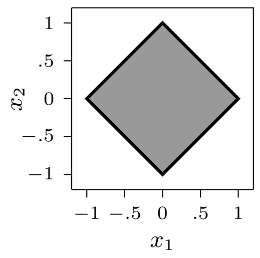

Figure 2.4: Sublevel set for the constraint of (2.3): $\{x \in \mathbb{R}^2 : \|x\|_1 \leq 1\}$.

It is less obvious that this is the case than for the two previous methods, but the solutions obtained include exact zeros, with lower values of t resulting in more zeros. A proof of this fact relies on the Karush-Kuhn-Tucker conditions and will be given later in the course, but Figure 2.5 gives some intuition as to why this is the case in $\mathbb{R}^2$.

The gray diamond represents the set of solutions allowable by the constraint on $\|x\|_1$, the black circle is the least-squares solution, and the contours are level sets of the objective $\|y - Ax\|^2$. The first contour to intersect the feasible set is the optimal solution. In this case, that corresponds to a sparse solution. In higher dimensions, the sparse edges of the feasible set are even "spikier" and so it is very likely that the solution is sparse.

This is known as the Least Absolute Shrinkage and Selection Operator (LASSO) method, and is effectively a convex relaxation of best subset selection. Thanks to the convexity, this problem can be solved readily, and many properties are known about the solutions; the proof of consistency, for example, follows readily from properties of convexity and the nature of the errors.

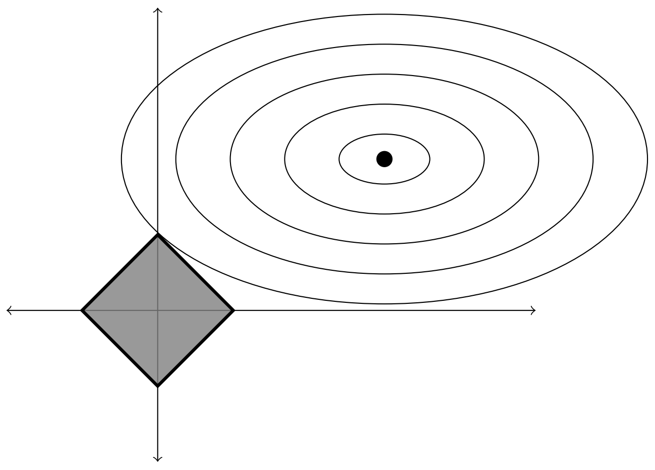

Figure 2.5: An illustration of why the lasso method gives solutions with zeros in $\mathbb{R}^2$.

2.2 Worked Example: Image Understanding

In this section we give another example for how to cast a task as an optimization problem. If we want the computer to understand the content of an image captured by a camera, how can we model that? We basically attempt to map the pixels of the image to some computational representation. Depending on the representation, there are different approaches to the problem:

- **Inverse Computer Graphics:** We can assume that the image is the rendering output of a 3D scene, which involves a set of transformations applied to a set of 3D objects; our goal is to recognize these objects and transformations. Rather than a full 3D approach, one call also use a 2.5D approach (a.k.a. the "pop-up" approach) where we model 2D textures along with depth coordinates.
- **2D Image Classification**: Instead, we can stick to the 2D nature of the image and identify it by finding the best match among images known a priori. A further step would be to segment the image and classify each segment.

We will focus on the 2D segmentation/classification model to discuss different aspects of this optimization problem. Basically, we want to define optimization variables and the optimization objective/constraints.

We do not distinguish between objective and constraints here because constraints can be formulated as penalty terms in the objective function. This is more obvious for *soft constraints*; constraints that, when violated, make the solution less optimal but not infeasible. Most of the constraints mentioned in this section are soft constraints.

2.2.1 Optimization Variable

Firstly, any optimization problem needs an object to be optimized. One possibility would be to define a segment ID for each pixel and a class ID for each segment. Another option is to define variables on edges between pixels rather than pixels themselves. In addition to these variables, there are many properties of the image that can be used to formulate the objective and constraints such as color, edges (detected by some edge detection algorithm), proximity relation, SIFT descriptors, and many more.

2.2.2 Constraints

Secondly, the optimization problem needs reasonable and well-defined constraints on the optimization variables, which affect the accuracy and complexity of the formulated problem. For example:

- Range constraint: the segment ID x should be within a fixed range $x \in \{1, 2, \cdots, 10\}$, which is used to control the number of partitions. A similar constraint holds for class ID.
- Proximity constraint: adjacent pixels should have same segment ID with high probability. The grid graph (cf. Figure 2.6) is one possible way to represent proximity constraints. Each vertex represents a pixel; each edge connects two adjacent pixels. Connected vertices should prefer to have the same segment ID. More generally, we can add edges with smaller weights connecting non-adjacent pixels to provide more fine-tuned control of similarity among all pixels.

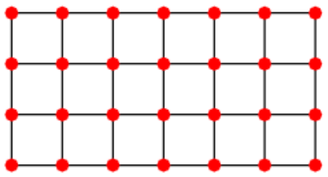

Figure 2.6: Grid Graph

- Edge constraint: pixels on different sides of an edge should prefer to have different segment IDs.
- Balancing constraint: the size of segments should be balanced (to avoid trivial segmentations, such as one corner pixel belonging to a segment and the rest of the image belongs to another segment).
- Smoothness constraint: a segment should not contain an abrupt color transition. This could be stated as a constraint that $\sum_{x_i, x_j \in E_i} \phi(x_i, x_j)$ should be close to 0, where E_i is the set of pairs of adjacent pixels in segment i and ϕ is a monotonic function of the difference in color.

2.2.3 Mechanism and Methods

As for solving the optimization problem itself, there are also different methods according to various optimization properties. For example, after formulating the problem, an exact solution might turn out to be intractable. In this case one can use an approximate version that is easier to solve, for example by relaxing integer or binary variables to be real variables. Having decided an appropriate approximation, there could be a number of algorithms to choose from, each with pros and cons. It is not uncommon to go back and forth between different decision steps: for example, considering your algorithmic options might cause you to adjust your approximation, and your approximation method might lead you to change the formulation to be more easily approximated.

2.3 Gradient Descent

Gradient descent is the grandfather of first order methods. It simply starts at an initial point and then repeatedly takes a step opposite to the gradient direction of the function at the current point. The gradient descent algorithm to minimize a function $f(x)$ is as follows:

```
**for** $k = 0, 1, 2, \ldots$ **do**
    $g_k \leftarrow \nabla f(x_k)$
    $x_{k+1} \leftarrow x_k - t_k g_k$
**end for**
```

Figure 2.7 shows gradient descent in action. The figure shows a function in two variables, a and b. The contours in the plot are *level sets*, where the value of the function is constant along the contour. Darker colors indicate lower values of the function. Note that the function has two minima: a local minimum to the right and a global minimum to the left.

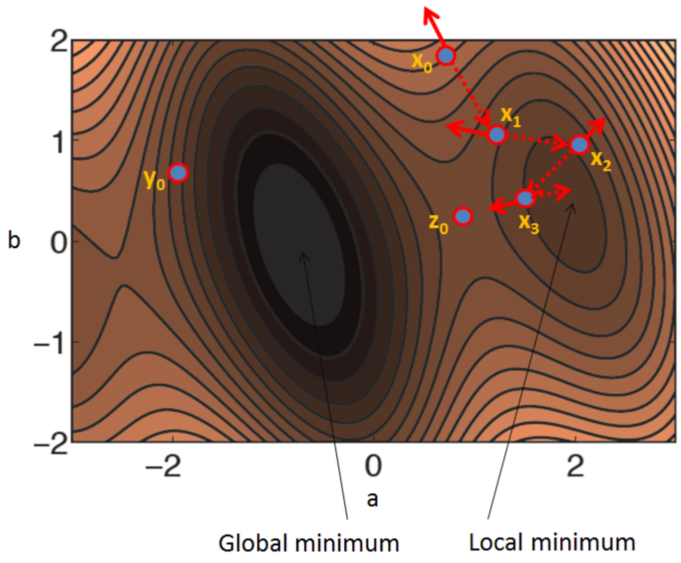

Figure 2.7: Demonstration of gradient descent.

The figure shows how the algorithm proceeds starting from x_0. At each point the gradient (solid arrow) is computed and a step is taken in the opposite direction (dotted arrow), until the algorithm stops at some point near the local minimum.

From the description of the algorithm and the figure, we can see that there are several major choices to be made in implementing gradient descent: the initial point x_0, the step size t_k, and the stopping condition. We will not discuss choice of t_k at this time.

2.3.1 Where Do We Start?

Ideally, one would choose x_0 to be close to the minimum. In many situations, however, it is very difficult to make such a guess; we often simply choose zero.

The demonstration in Figure 2.7 demonstrates that for some functions, the choice of initial point may be critical. For example, our choice of x_0 resulted in finding a local minimum. Had we started at a point like y_0, we would have reached the global minimum, the actual correct answer. We also might be extremely unlucky and start precisely at z_0. At z_0 the gradient is 0, because this point is minimum with respect to a and maximum with respect to b (see how the color changes along the vertical and horizontal directions).

Such a point is called a *saddle point*, as it resembles the saddle of a horse. If we start at z_0 the algorithm will immediately terminate.

A common workaround for this problem is to run the algorithm several times from widely dispersed starting points, and then pick the best result.

2.3.2 When Do We Stop?

When we stop iterating depends on the problem setting. Ideally, we stop when $f(x_k)$ is "close enough" to the minimum. What is "close enough" depends on the situation at hand.

Early Stopping

Held-out Data: In machine learning and statistics, we sometimes want to employ *early stopping* to avoid overfitting. We often wish to optimize a sample average based on given data. Specifically, if x denotes the parameters we are optimizing and i denotes a data point from a distribution $p(i)$, then we usually try to minimize $f(x) = \mathbb{E}_{p(i)}[f_i(x)]$ where f_i is the portion of the function corresponding to data point i. If we have an i.i.d. sample $I \sim p(i)$, we can then simply optimize the sample mean $\hat{f}(x) = \sum_{i \in I} f_i(x)/|I|$.

For example, in linear regression we try to minimize the squared error loss function $f(x) = \mathbb{E}_{a,b}|a \cdot x - b|^2$, where $a \in \mathbb{R}^p$ is a feature vector, $b \in \mathbb{R}$ is the true output prediction and $x \in \mathbb{R}^p$ is the weight vector we are optimizing. $f_i(x)$ is then simply $|a_i \cdot x - b_i|^2$.

Fully optimizing this sample mean might overfit the data, resulting in poor generalizability to new data. A common solution to this problem is to evaluate the objective function on a held-out validation sample, and stop when the objective begins to increase on that sample, a sign of overfitting. This process is demonstrated in Figure 2.8. This way, early stopping acts as a sort of regularization.

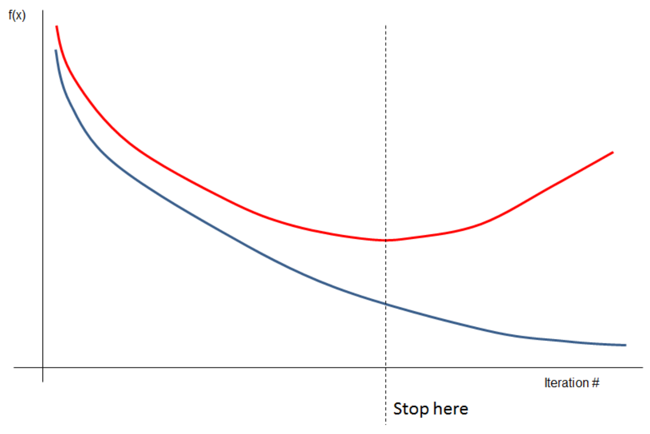

Figure 2.8: Early stopping for regularization. The figure shows the value of the objective function on training data (blue) and held-out data (red) vs the number of steps. The black line indicates where to stop, when the objective function begins to increase on held-out data.

When it is unnecessary to proceed: In some other situations, the problem setting limits the effective (or desired) precision of the solution. For example, if we know that randomness of the data limits our confidence in the solution to two decimal places, there is no point in spending significant amounts of computation optimizing x to seven decimal places.

Using Convergence Bounds

Sometimes we have no reason for early stopping, and wish to ensure that $f(x_k)$ is close enough to the minimum in a principled way. In this case, to estimate the minimum number of steps, we resort to convergence bounds, which take the form

$$K \geq K_f \; (f(x_0) - f(x^*)) \; g\left(\tfrac{1}{\varepsilon}\right),$$

where K is the number of steps, K_f is a parameter that can be estimated from the properties of the function, g is function that also depends on the properties of the function, ε is the error tolerance, and $f(x^*)$ is the true minimum, which can be estimated from duality. More details on that will be revealed later in the course.

2.3.3 Other Issues

In addition to local minima and the (rare) possibility of getting stuck at a saddle point, there are other issues that should be taken into consideration.

For example, if the step size t_k remains large, it may lead to an oscillatory behavior that does not converge.

Another issue is that, depending on the function and starting point, gradient descent could continue indefinitely because there is no minimum. Consider for example minimizing e^x: there is no finite x for which $\frac{d}{dx}e^x = 0$. Other functions could have such asymptotic minima but also a global minimum of lower value; gradient descent, depending on its starting point, might forever chase the asymptote, unaware of the true answer elsewhere in the search space.