Stat 8054 Lecture Notes: Slack Variables

Charles J. Geyer

December 22, 2023

1 License

2 R

- The version of R used to make this document is 4.3.2.
- The version of the `glpkAPI` package used to make this document is 1.3.4.
- The version of the `Matrix` package used to make this document is 1.6.4.
- The version of the `rmarkdown` package used to make this document is 2.25.

```
library(Matrix)
library(glpkAPI)
```

```
## using GLPK version 4.65
```

3 Motivation

A student asked in class discussion "Are there analogs of isotonic regression to regressing on the median or other quantiles?"

4 Slack Variables

A "slack variable" is a nonnegative variable introduced to allow functions with discontinuous derivatives to be represented in the form required by common optimization software. For example, the absolute value function can be represented by replacing one variable with two. Instead of $|x|$ we write $x_1 + x_2$ where $x = x_1 - x_2$ and both x_1 and x_2 are required to be nonnegative.

5 Isotonic L1 Regression

We take for our example a simplified version of the isotonic regression problem where there are no repeated predictor values. This can be generalized to handle repeated predictor values as discussed in the isotonic regression handout.

$$\begin{aligned} &\text{minimize} \sum_{i=1}^{n} |y_i - \mu_i| \\ &\text{subject to } \mu_1 \le \mu_2 \le \cdots \le \mu_n \end{aligned}$$

And we reformulate this using slack variables in the obvious way (there may be a nonobvious way to reformulate it using only half as many slack variables but I cannot be bothered to think it up).

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{n} (u_i + v_i) \\ \text{subject to} \quad & u_i \geq 0, && i = 1, \ldots, n \\ & v_i \geq 0, && i = 1, \ldots, n \\ & y_i - \mu_i = u_i - v_i, && i = 1, \ldots, n \\ & \mu_i \leq \mu_{i+1} && i = 1, \ldots, n - 1 \end{aligned}$$

6 Try It

6.1 Data

Make up some data

```
set.seed(42)
n <- 30
x <- seq(1, 3, length = 30)
y <- rexp(n, 1 / x)
```

6.2 Linear Programming

Now we set up our problem as linear programming, as discussed in the preceding section. Our variables are u_i, v_i, and μ_i for $i = 1, \ldots, n$ but linear programming treats them as components of one state vector. So we have $3n$ variables n equality constraints and $3n - 1$ inequality constraints.

The linear programming software we are going to use allows solves problems of the form

$$\begin{aligned} \text{minimize} \quad & g^T x \\ \text{subject to} \quad & a \leq x \leq b \\ & c \leq Mx \leq d \end{aligned}$$

where x is the variable which we are optimizing and everything else is a constant (a, b, c, and d are vectors and M is a matrix). The lower bound vectors a and c can have components set to `-Inf` to allow for unbounded variables and the upper bound vectors b and d can have components set to `Inf` to allow for unbounded variables. Equality constraints are expressed in the scheme above by setting the lower bound and upper bound the same.

We make this specific for our problem as follows

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{n} (u_i + v_i) \\ \text{subject to} \quad & -\infty \leq \mu_i < \infty, && i = 1, \ldots, n \\ & 0 \leq u_i \leq \infty, && i = 1, \ldots, n \\ & 0 \leq v_i \leq \infty, && i = 1, \ldots, n \\ & y_i = \mu_i + u_i - v_i, && i = 1, \ldots, n \\ & 0 \leq \mu_{i+1} - \mu_i \leq \infty && i = 1, \ldots, n - 1 \end{aligned}$$

So among the $3n - 1$ inequality constraints we have $2n$ bound constraints and $n - 1$ general inequality constraints.

We will have the variables (components of x) be the μ_i, the u_i, and the v_i in that order.

```
gg <- c(rep(0, n), rep(1, 2 * n))
aa <- c(rep(-Inf, n), rep(0, 2 * n))
bb <- rep(Inf, 3 * n)
cc <- c(y, rep(0, n - 1))
dd <- c(y, rep(Inf, n - 1))
idmat <- diag(n)
mm <- cbind(idmat, idmat, - idmat)
mm2 <- matrix(0, n - 1, 3 * n)
mm2[row(mm2) == col(mm2)] <- -1
mm2[row(mm2) + 1 == col(mm2)] <- 1
mm <- rbind(mm, mm2)
```

Now we are set up, math objects g, a, b, c, d, and M are R objects `gg`, `aa`, `bb`, `cc`, `dd`, and `mm` (we doubled the letters to avoid the name of R function `c`).

```
lp <- initProbGLPK()
addRowsGLPK(lp, nrow(mm))
```

```
## [1] 1
```

```
addColsGLPK(lp, ncol(mm))
```

```
## [1] 1
```

```
# STFU
setSimplexParmGLPK(MSG_LEV, GLP_MSG_OFF)
# column bounds: free variables
idx <- which(aa == -Inf & bb == Inf)
setColsBndsGLPK(lp, idx, aa[idx], bb[idx], rep(GLP_FR, length(idx)))
# column bounds: other variables
idx <- which(aa == 0 & bb == Inf)
setColsBndsGLPK(lp, idx, aa[idx], bb[idx], rep(GLP_LO, length(idx)))
# row bounds: fixed variables
idx <- which(cc == dd)
setRowsBndsGLPK(lp, idx, cc[idx], dd[idx], rep(GLP_FX, length(idx)))
# row bounds: other variables
idx <- which(cc < dd)
setRowsBndsGLPK(lp, idx, cc[idx], dd[idx], rep(GLP_LO, length(idx)))
# objective function
setObjDirGLPK(lp, GLP_MIN)
idx <- seq_along(gg)
setObjCoefsGLPK(lp, idx, gg)
# constraint matrix
mm <- Matrix(mm)
qux <- mat2triplet(mm)
idx <- qux$x != 0
loadMatrixGLPK(lp, sum(idx), qux$i[idx], qux$j[idx], qux$x[idx])
```

The types of variables are

- `GLP_FR` free, unbounded in both directions,
- `GLP_FX` fixed, equal to the lower bound (upper bound ignored),
- `GLP_LO` lower bound only (no upper bound),
- `GLP_UP` upper bound only (no lower bound), and
- `GLP_DB` double bound (lower bound and upper bound both used).

Also note that this packages uses sparse arithmetic for the constraint matrix. If we had used sparse arithmetic

all the way through (using R package `Matrix`), then we could have done a very large problem with millions of variables without using more memory than in a laptop. The fact that the constraint matrix has $3n(n-1)$ components would not matter. It has only $3n + 2(n-1)$ nonzero components.

Now we have loaded the data into the format required by this package. And we are ready to solve.

```
solveSimplexGLPK(lp)
```

```
## [1] 0
```

```
getPrimStatGLPK(lp) == GLP_FEAS
```

```
## [1] TRUE
```

`TRUE` means solution status optimal.

```
x <- getColsPrimGLPK(lp)
```

Only the first n components of `x` are interesting.

```
mu <- x[1:n]
```

Let's take a look.

```
plot(y)
points(mu, pch = 19)
```

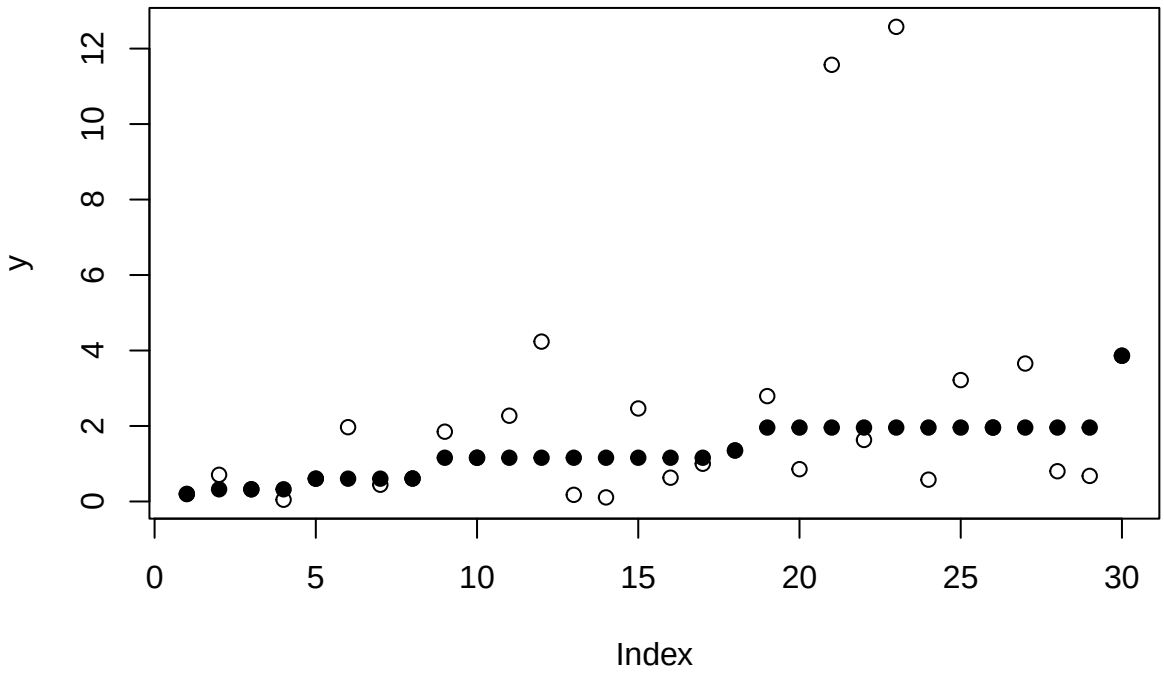

Figure 1: Scatterplot with Isotonic L1 Regression. Hollow dots observed, solid dots fitted.

6.3 Kuhn-Tucker Conditions

6.3.1 Primal Feasibility

First get the rest of the variables.

```
u <- x[(n + 1):(2 * n)]
v <- x[(2 * n + 1):(3 * n)]
```

And check the constraints on them.

```
all(u >= 0)
```

```
## [1] TRUE
```

```
all(v >= 0)
```

```
## [1] TRUE
```

```
all(diff(mu) >= 0)
```

```
## [1] TRUE
```

```
all(y == mu + u - v)
```

```
## [1] FALSE
```

```
max(abs(y - (mu + u - v)))
```

```
## [1] 6.661338e-16
```

OK. Primal feasibility checks, to within the accuracy of computer arithmetic. Note that the test failed when we naively expected the computer to have real real numbers. But it is clear that $6.6613381 \times 10^{-16}$ is just rounding error (inexactness of computer arithmetic).

We might have had to make similar modifications to the other tests, but the naive approach just happened to work.

6.3.2 Dual Feasibility

Now get Lagrange multipliers. These are called dual variables by the linear programming software.

```
lambda.row <- getRowsDualGLPK(lp)
lambda.col <- getColsDualGLPK(lp)
length(lambda.row) == nrow(mm)
```

```
## [1] TRUE
```

```
length(lambda.col) == ncol(mm)
```

```
## [1] TRUE
```

We are not sure what sign conventions the linear programming software is using, so we just look at what we got.

```
lambda.col.mu <- lambda.col[1:n]
lambda.col.u <- lambda.col[(n + 1):(2 * n)]
lambda.col.v <- lambda.col[(2 * n + 1):(3 * n)]
range(lambda.col.mu)
```

```
## [1] 0 0
```

```
range(lambda.col.u)
```

```
## [1] 0 2
```

```
range(lambda.col.v)
```

```
## [1] 0 2
```

Actually the `lambda.col.mu` are not Lagrange multipliers because they do not correspond to constraints (the μ_i are unbounded).

This checks if the rule is that Lagrange multiplers for lower bound constraints are nonnegative. We have no upper bound constraints in this part.

```
nrow(mm) == 2 * n - 1
```

```
## [1] TRUE
```

```
lambda.row.y <- lambda.row[1:n]
lambda.row.mu <- lambda.row[- (1:n)]
range(lambda.row.mu)
```

```
## [1] 0 3
```

```
range(lambda.row.y)
```

```
## [1] -1  1
```

The check for `lambda.row.y` is OK. These are for the equality constraints $y = \mu + u - v$, and Lagrange multipliers for equality constraints can be either sign.

The way we wrote the μ constraints $\mu_{i+1} - \mu_i \geq 0$, these are again lower bound constraints, so if the rule we guessed above is correct, then these are also correct (all nonnegative).

Since we have no upper bound constraints in this part either, we are not going to learn the rule for upper bound constraints, unless we rewrite the problem.

6.3.3 Complementary Slackness

```
range(lambda.col.mu * mu)
```

```
## [1] 0 0
```

```
range(lambda.col.u * u)
```

```
## [1] 0 0
```

```
range(lambda.col.v * v)
```

```
## [1] 0 0
```

```
range(lambda.row.y * (y - (mu + u - v)))
```

```
## [1] -6.661338e-16  4.440892e-16
```

```
range(lambda.row.mu * diff(mu))
```

```
## [1] 0 0
```

Everything checks. All zero up to accuracy of computer arithmetic.

6.3.4 Lagrangian Derivative Zero

The gradient vector of the objective function is `gg` so the derivative evaluated at the solution is

```
sum(gg * x)
```

```
## [1] 40.35924
```

After some trial and error, we decide the Lagrangian is

$$\mathcal{L} = \left[\sum_{i=1}^{n} (u_i + v_i) \right] - \lambda_u^T u - \lambda_v^T v - \lambda_y^T (\mu + u - v - y) - \sum_{i=1}^{n-1} \lambda_\mu(i)(\mu_{i+1} - \mu_i)$$

and the derivatives are

$$\begin{aligned} \partial \mathcal{L} / \partial \mu_i &= -\lambda_y(i) - \lambda_\mu(i-1) + \lambda_\mu(i) \\ \partial \mathcal{L} / \partial u_i &= 1 - \lambda_u(i) - \lambda_y(i) \\ \partial \mathcal{L} / \partial v_i &= 1 - \lambda_v(i) + \lambda_y(i) \end{aligned}$$

where we define $\lambda_\mu(0) = \lambda_\mu(n) = 0$ in order not to have to special case the first equation for $i = 1$ and $i = n$.

So the checks are

```
range(- lambda.row.y - c(0, lambda.row.mu) + c(lambda.row.mu, 0))
```

```
## [1] 0 0
```

```
range(1 - lambda.col.u - lambda.row.y)
```

```
## [1] 0 0
```

```
range(1 - lambda.col.v + lambda.row.y)
```

```
## [1] 0 0
```

So everything checks. We have proved this is the unique global optimum (because this is a convex problem).