Linear Models with Julia

(Scratching the Surface...)

Shane Conway (smc77@columbia.edu)

May 1, 2012

Linear regression is one of the most useful tools in Statistical Computing.

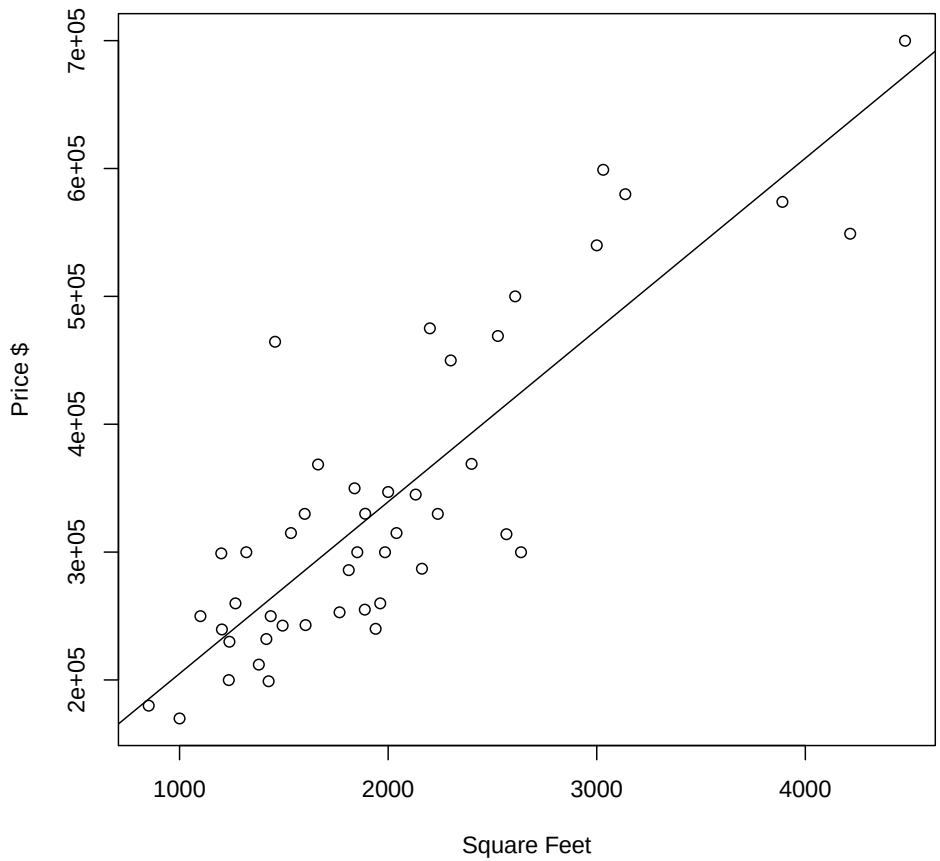
$$\hat{y} = f(X, \beta) = \beta X \tag{1}$$

With OLS, the task is to minimize the residuals:

$$\hat{\beta} = \mathit{argmin}||y - \beta X||^2 \tag{2}$$

Julia currently has a little support for OLS with the linreg function (using the normal equations in linalg.jl).

Example: Predict housing price based on number of bedrooms and square footage (taken from mlclass.org).



R was designed to handle these kinds of problems ("Programming with Data"):

```
data = read.csv("housing.txt")
colnames(data) <- c("sqr.feet", "rooms", "price")
housing.lm = lm(price ~ sqr.feet, data=data)
plot(data[,3] ~ data[,1], xlab="Square Feet",
ylab="Price \$")
abline(housing.lm)
summary(housing.lm)
```

R has sophisticated tools for computing OLS.

- lm() originally written by John Chambers while at Bell Labs for S language.
- formula's (e.g. $y \sim x1 + x2$) introduced in Chambers, J. M. and Hastie, T. J. (1992) "Statistical Models in S", based on Wilkinson and Rogers (1973) "Symbolic descriptions of factorial models for analysis of variance."
- Reimplemented in R by Ross Ihaka.

lm() works with many related functions such as plot, summary, and predict. And it provides a model for other statistical functions (e.g. predict.glm, predict.nls...).

Many modern statistical methods (e.g. bootstrapping, cross-validation) require repeated fitting, so small performance differences matter.

R's lm() function uses a pivoted QR algorithm with LINPACK.

Doug Bates, Dirk Eddelbuettel, and Romain Francois have made further improvements to performance in RcppArmadillo and RcppEigen. (see http://dirk.eddelbuettel.com/blog/2011/07/05/ for more detail). Also see Doug Bates "Least Squares Calculations in R: Timing different approaches" in "R News 2004, 1".

OLS can be fit based on variety of different methods, including:

- Direct
  - Normal Equations ($\beta = (X'X)^{-1}X'y$)
  - SVD decomposition
  - QR decomposition
- Iterative
  - Gradient descent
  - Newton's method
  - Conjugate gradient

Iterative methods can be better for very large data and can be applied to nonlinear problems.

We can easily compute this directly using routines from LAPACK:

```
data = csvread("housing.txt");
X = data[:, 1:2];
y = data[:, 3];

# Normal equations
beta = inv(X'*X) * X' * y

# QR decomposition
(m, n) = size(X);
(Q, R) = qr(X);
beta = R[1:n, :] \ (Q' * y)
```

Or iteratively using Gradient Descent:

```
function cost(X, y, theta)
J = 1 / (2 * length(y)) * (X * theta - y)' * (X * theta - y
return J[1]
end

function gradient_descent(X, y, theta, alpha, num_iters)
m = length(y);
J_history = zeros(num_iters, 1);
for iter = 1:num_iters
h_theta = X * theta;
d = (h_theta - y)' * X;
theta = theta - (alpha * 1/m * d');
J_history[iter] = cost(X, y, theta);
end
return theta, J_history
end
```

Standardizing the inputs makes the optimization more likely to converge.

```
function zscore(X)
mu = mean(X, 1);
sigma = amap(std, X, 2) # don't know how to apply std to co
transform = ones(size(X, 1), 1);
X = (X - transform * mu) ./ (transform * sigma');
return X, mu, sigma
end
```

```
m = length(y);
(X_norm, mu, sigma) = zscore(X);

X = [ones(m, 1) X];

# This fails: linreg(X, y)

alpha = 0.03;
num_iters = 1500;

theta = zeros(size(X, 2), 1);

(theta, J_history) = gradient_descent(X, y, theta, alpha, n
```

We can also compute the R^2 for the given predicted values.

$$TSS = \sum_i (y_i - \bar{y})^2 \tag{3}$$

$$RSS = \sum_i (y_i - \hat{y}_i)^2 \tag{4}$$

$$R^2 = 1 - \frac{TSS}{RSS} \tag{5}$$

Using Julia as:

```
tss = sum((y - mean(y)).^2)
rss = sum((y - (X * beta)).^2)
r_2 = 1 - (rss/tss)
```