15.095 Homework 1

Kim-Anh-Nhi Nguyen

2018-09-19

1 Modeling Furniture Ordering

a Mixed-Integer linear optimization formulation

Jack's goal is to minimize the cost of his furniture purchase by selecting the optimum numbers of sets he is going to buy from the optimal companies.

For each company, he should know:

- Whether he will buy from it or not
- If he does, the number of sets of furniture he will buy from that company.

So, let us introduce the following variables:

- $x_i \in \{0, 1\}$: binary variable for company n. i, equal to 1 if Jack buys from company n. i, equal to 0 else.
- c_i: integer variable for the number of sets Jacks buys from company n. i.

Where the indices used are corresponding to the companies:

Company number	Company name
1	Carolina Woodworks
2	Nashawtuc Millworks
3	Adirondack Furnishing Designs
4	Lancaster Artisan Company
5	Delaware Mills

So, all variables are integers.

The cost function that we want to minimize is the total cost of the furniture purchase will be the sum of purchases from each company:

$$cost_{old} = (2500c_1 + 10000x_1) + (2450c_2 + 20000x_2) + (2510c_3) + (2470c_4 + 13000x_4)$$

The constraints are the following:

- Given M big ($M = 10000$ for example):

$$\forall i \in 1, 4, -Mx_i \leq c_i \leq Mx_i$$

We use the big-M method to force any $c_i = 0$ whenever $x_i = 0$

- Quantity constraint from company 1 (put name):

$$0 \leq c_1 \leq 1000$$

- Quantity constraint from company 2:

$$0 \leq c_2 \leq 1200$$

- Quantity constraint from company 3:

$$0 \leq c_3 \leq 800$$

- Quantity constraint from company 4:

$$0 \leq c_4 \leq 1100$$

- Quantity constraint from company 4:

$$0 \leq c_4 \leq 1100$$

- All 2,000 offices must be provided with furniture:

$$\sum_{i=1}^{4} c_i = 2000$$

b Mixed-Integer linear optimization formulation

The extra constraints we need to add in the following conditions are:

i Jack must order from at least 3 companies

$$\sum_{i=1}^{4} x_i = 3$$

ii Jack can order from Carolina Woodworks or Nashawtuc Millworks, but not both.

$$x_1 + x_2 \leq 1$$

iii If Jack orders from Carolina Woodworks, he must also order from Nashawtuc Mill works. However, if Jack orders from Nashawtuc Millworks, he may or may not order from Carolina Woodworks.

Subscript:

$$x_2 - x_1 \geq 0$$

iv Jack can either order from both Carolina Woodworks and Nashawtuc Millworks or from neither.

Subscript:

$$x_2 = x_1$$

v If Jack does not order from Carolina Woodworks, then he must order from Nashawtuc Millworks

Subscript:

$$x_2 + x_1 \geq 1$$

c Incorporate a fifth company

For the fifth company, the cost of purchase depends on the number of furniture sets that are bought.

We introduce 4 new variables to take the 2 situations into account:

- c_5: integer variable for the number of sets Jacks buys from company n.5 when this number is strictly less than 1,000
- c_6: integer variable which represents the number of sets that exceeds 1,000, when it happens
- x_5: binary variable, equals to 1 when the number of furniture sets bought from company 5 is strictly less than 1,000
- x_6: binary variable, equals to 1 when the number of furniture sets bought from company 5 is greater than 1,000

The new total cost that we want to minimize becomes the following:

$$cost_{new} = cost_{old} + 2530c_5 + 9000x_5 + 2430c_6 + 7000x_6$$

To the previous minimization problem, we add the following constraints:

- Quantity limit: $0 \leq c_5 \leq 1000$
- Quantity limit: $0 \leq c_6 \leq 500$
- Given M big ($M = 10000$ for example):

  $$\forall i \in 5, 6, -Mx_i \leq c_i \leq Mx_i$$

  We again, use the big-M method to force any $c_i = 0$ whenever $x_i = 0$
- $0 \leq c_5 + c_6 \leq 1500$
- If Jack buys more than 1,000 sets from company 5, then $x_6 = 1$ and $x_5 = 1$. If he buys less than 1,000 sets, then $x_6 = 0$ and $x_5 = 1$. If he doesn't buy from company 5, then $x_6 = 0$ and $x_5 = 0$. So the constraint that satisfies these conditions is:

  $$x_5 - x_6 \geq 0$$

- Moreover, if $x_6 = 1$, i.e. he buys more than 1,000 sets from company 5, then $c_5 = 1000$. The constraint satisfying this condition can be:

$$c_5 \geq 1000 - M(1 - x_6)$$

  and

$$c_5 \leq 1000 + Mx_6$$

  So that, when $x_6 = 1$, $c_5 = 1000$, and when $x_6 = 0$, $c_5 \leq 1000$

- And, we change the $\sum_{i=1}^{4} c_i = 2000$ to

$$\sum_{i=1}^{6} c_i = 2000$$

2 Robust Linear Regression

a Julia/JuMP solutions

Cf. notebook at the end of the document in the Appendices (part 4 of the homework). There are 2 appendices:

- Appendix a is the notebook where the models where trained and tested on the data set `housing.csv`
- Appendix b is the same notebook, using the data set `communities-and-crime.csv`

In the notebooks, we have used 3 different functions to train the model according to 3 different linear regression techniques:

- `standardlinear`: function to train a standard linear regression. We want to find:

$$\min_{x} ||y - X\beta||_2$$

  This can be written as a linear optimization problem as:

$$\min_{x} t$$

$$\text{s.t. } ||y - X\beta||_2 \leq t$$

- `lassolinear`: function to train a l_1-regularized linear regression, also called Lasso method. We want to find:

$$\min_{\beta} ||y - X\beta||_2 + \rho||\beta||_1$$

  This can be written as a linear optimization problem as:

$$\min(t + \rho \sum_{i=1}^{p} a_i)$$

$$\text{s.t.}$$

$$||y - X\beta||_2 \leq t$$

$$\forall i, a_i \geq \beta_i$$

$$\forall i, a_i \geq -\beta_i$$

- `ridgelinear`: function to train a l_2-regularized linear regression, also called Ridge method. We want to find:

$$\min_{\beta} ||y - X\beta||_2 + \rho||\beta||_2$$

  This can be written as a linear optimization problem as:

$$\min{(t + \rho u)}$$

$$\text{s.t.}$$

$$||y - X\beta||_2 \leq t$$

$$||\beta||_2 \leq u$$

b Linear regression fitting

For l_1-regularization and l_2-regularization linear regression, we need to find the best value of ρ first.
In the notebooks, we used a function `findBestRho` which finds the best value of ρ by solving the optimization problem on the training set for each different value of ρ. Then we take the value of ρ for which the score $||y_{val} - X_{val}\beta^*||$ is the lowest on the validation set.

Here is a summary of the optimal values of ρ found for each dat aset and each robust linear regression method:

Linear regression method	`housing.csv` dataset	`communities-and-crime.csv` dataset
l_1-regularization (Lasso)	$\rho = 0.1$	$\rho = 1.0$
l_2-regularization (Ridge)	$\rho = 1.0$	$\rho = 2.0$

Then, the second step consist in, using the optimal ρ found before, to find the optimal value of β for each technique, including standard linear regression, using the training and the validation set. We also performed a standard linear regression. (cf. appendices to see the values pf beta found).

Then, in the third step, we computed $score = ||y_{test} - X_{test}\beta^*||_2$ on the testing set using the optimal values of β to score each regression method.

Lastly, to compare the performance between the 3 different linear regression, we compute the baseline model which computes the means $\hat{y}_{train+val}$ on the training and validation set.
Here is a summary of the scores found for each data set and each linear regression method:

Linear regression method	`housing.csv` data set	`communities-and-crime.csv` data set
standard regression	$score = 91.32$	$score = 2.548$
l_1-regularization (Lasso)	$score = 89.47$	$score = 0.78$
l_2-regularization (Ridge)	$score = 83.60$	$score = 0.75$
Baseline	$score = 129.23$	$score = 1.46$

To assess the performance of each linear regression model on each data set, we compare the score of the chosen algorithm and the score of the baseline model.

We compute the relative improvement on scores: $\frac{(score_{model} - score_{baseline})}{score_{baseline}}$

Linear regression method	Relative improvement to baseline with `housing.csv`	Relative improvement to baseline with `communities-and-crime.csv`
standard regression	+29%	−75%
l_1-regularization (Lasso)	+31%	+46%
l_2-regularization (Ridge)	+35%	+49%

c Benefits

Comments about the performances:

- Thanks to the last table, we can notice that ridge regression performs the best on both data sets and standard linear regression performs the worse on both data sets.
- On `housing.csv`, all 3 models make better predictions than the baseline model, at approximately the same accuracy: the improvement is of $\sim 30\%$
- On `communities-and-crime.csv`, both robust linear regression models make much better predictions ($> 45\%$) than the baseline model and the standard linear regression. The latter does even much worse than the baseline model (-75%)
- These big performance differences between models depending on data sets can be explained by the structure of each data set: `housing.csv` has a relatively high number of data points (506 rows, training was done on 252 points) and a low number of variables (13 columns), whereas `communities-and-crime.csv` has a relatively low number of data points (122 rows but training was done on 60 points) and a very high number of variables (122 columns).

Conclusions:

- Therefore, it seems that regularized linear regression performs much better and has benefits when the training is don on a data set where the number of variables is large and the number of data points is small. It should be used over standard linear regression in those cases.
- The benefits of regularized linear regression is that it predicts well the uncertainty/variability around the data: it predicts well even with small changes on the data.

3 Best Subset Selection

In this problem, we want to:

$$\min_{\beta} f(\beta)$$

$$\text{s.t. } ||\beta||_0 \leq k$$

where $f(\beta) = ||y - X\beta||_2^2 + \Gamma||\beta||_1$ and $\Gamma > 0$

We use the same method as presented in Lecture 2.

Using the fact that $g(\beta) = ||y - X\beta||_2^2$ is convex and has l-Lipschitz gradient we had conclude that:

$$\forall L \geq l : g(\beta) \leq \frac{L}{2}||\beta - u||_2^2 - \frac{1}{2L}||\nabla g(\beta_0)||_2^2$$

where $u = \beta_0 - \frac{\nabla(\beta_0)}{L}$

So:

$$f(\beta) \leq \frac{L}{2}||\beta - u||_2^2 - \frac{1}{2L}||\nabla g(\beta_0)||_2^2 + \Gamma||\beta||_1$$

$-\frac{1}{2L}||\nabla g(\beta_0)||_2^2$ is a constant relative to β, so our minimization problem can be solved by solving the following optimization:

$$\min_{\beta} R(\beta)$$

$$\text{s.t. } ||\beta||_0 \leq k$$

$$\text{where } R(\beta) = \frac{L}{2}||\beta - u||_2^2 + \Gamma||\beta||_1$$

$R(\beta) = \frac{L}{2}\sum_{i=1}^{p}(\beta_i - u_i)^2 + \sum_{i=1}^{p}|\beta_i|$

So minimizing $S(\beta_i) = (\beta_i - u_i)^2 + |\beta_i|$ for each component i is sufficient to minimize $R(\beta)$.

Then, we can simplify $S(\beta_i)$ by separating it into 2 cases.

Case 1: $\beta_i \geq 0$

Then:

$$S(\beta_i) = \frac{L}{2}(\beta_i^2 - 2\beta_i u_i + u_i^2) + \Gamma\beta_i$$

$$S(\beta_i) = \frac{L}{2}[\beta_i^2 - 2\beta_i u_i + u_i^2 + \frac{2\Gamma}{L}\beta_i]$$

$$S(\beta_i) = \frac{L}{2}[\beta_i^2 - 2\beta_i(u_i - \frac{\Gamma}{L}) + u_i^2]$$

$$S(\beta_i) = \frac{L}{2}[\beta_i^2 - 2\beta_i(u_i - \frac{\Gamma}{L}) + u_i^2 - 2u_i\frac{\Gamma}{L} + 2u_i\frac{\Gamma}{L} + (\frac{\Gamma}{L})^2 - (\frac{\Gamma}{L})^2]$$

$$S(\beta_i) = \frac{L}{2}[[\beta_i - (u_i - \frac{\Gamma}{L})]^2 + 2u_i\frac{\Gamma}{L} - (\frac{\Gamma}{L})^2]$$

$$S(\beta_i) = \frac{L}{2}[\beta_i - (u_i - \frac{\Gamma}{L})]^2 + u_i\Gamma - \frac{\Gamma^2}{2L}$$

The term $u_i\Gamma - \frac{\Gamma^2}{2L}$ is a constant relative to β
So, minimizing $S(\beta_i)$ us equivalent to minimizing $T_1(\beta_i) = [\beta_i - (u_i - \frac{\Gamma}{L})]^2$ (we ignore the multiplicative factor $\frac{L}{2}$ as it is positive).

__Case 2__: $\beta_i \leq 0$
Then:

$$S(\beta_i) = \frac{L}{2}(\beta_i^2 - 2\beta_i u_i + u_i^2) - \Gamma\beta_i$$

Using the same method as for Case 1, we get:

$$S(\beta_i) = \frac{L}{2}[\beta_i - (u_i + \frac{\Gamma}{L})]^2 - u_i\Gamma - \frac{\Gamma^2}{2L}$$

The term $-u_i\Gamma - \frac{\Gamma^2}{2L}$ is a constant relative to β
So, minimizing $S(\beta_i)$ us equivalent to minimizing $T_2(\beta_i) = [\beta_i - (u_i + \frac{\Gamma}{L})]^2$ (we ignore the multiplicative factor $\frac{L}{2}$ as it is positive).

Now, how do we choose the best β_i in our first-order method?
There are 4 different values for β_i based on our previous calculations:

- If $u_i - \frac{\Gamma}{L} \geq 0$: then the new value for β_i can be $\beta_i = u_i - \frac{\Gamma}{L})$
- If $u_i - \frac{\Gamma}{L} \leq 0$: then the new value for β_i can only be $\beta_i = 0$ because we have seen that $u_i - \frac{\Gamma}{L})$ is applicable if only $\beta_i \geq 0$
- If $u_i + \frac{\Gamma}{L} \leq 0$: then the new value for β_i can be $\beta_i = u_i + \frac{\Gamma}{L})$
- If $u_i + \frac{\Gamma}{L} \geq 0$: then the new value for β_i can only be $\beta_i = 0$ because we have seen that $u_i + \frac{\Gamma}{L})$ is applicable if only $\beta_i \leq 0$

This has to be done for all components. Then, to satisfy the constraint $||\beta||_0 \leq k$, we can select the k largest components and set the others to 0.

There, here is the first-order algorithm we can implement:

Input: $f(\beta)$, Γ, L, ϵ

Output: A first order stationary solution β^*

- **Step 1**: Initialize $\beta_1 \in \mathbb{R}^p$ such that $||\beta||_0 \leq k$
- **Step 2**: For $m \geq 1$,
$$\beta_{m+1} \in H'_k(\beta_m)$$
- **Step 2**: Repeat Step 2, until $f(\beta_m) - f(\beta_{m+1}) \leq \epsilon$

The function H'_k does the following for β_m:

$$\forall i \in [\![1; p]\!]$$

Given $u_{m,i} = \beta_{m,i} - \frac{1}{L}\nabla(\beta_m)_i$

- Compute $u_{m,i} - \frac{\Gamma}{L}$.
If it is positive, set: $\beta^1_{m+1,i} = u_{m,i} - \frac{\Gamma}{L}$.
Otherwise, set: $\beta^1_{m+1,i} = 0$
- Compute $u_{m,i} + \frac{\Gamma}{L}$.
If it is negative, set: $\beta^2_{m+1,i} = u_{m,i} + \frac{\Gamma}{L}$.
Otherwise, set: $\beta^2_{m+1,i} = 0$
- compute $Argmin_{\beta^1_{m+1,i},\beta^2_{m+1,i}}[S(\beta^1_{m+1,i}), S(\beta^2_{m+1,i})]$: we keep the value of $\beta_{m+1,i}$ which gives the smallest value of the function S to minimize it.
- After setting all values for $(\beta_{m+1,i})_i$, select the k largest components $\beta_{m+1,i}$ and set the others to 0.

4 Appendices

a Notebook 1: using `housing.csv`

```
In [1]: using JuMP, Gurobi, DataFrames
```

```
In [2]: df = readtable("housing.csv",header=false)
        X = Matrix(df[1:end - 1])
        y = df[end];
```

```
WARNING: readtable is deprecated, use CSV.read from the CSV package instead
Stacktrace:
 [1] depwarn(::String, ::Symbol) at .\deprecated.jl:70
 [2] #readtable#232(::Bool, ::Char, ::Array{Char,1}, ::Char, ::Array{String,1}, ::Array{String,1
 [3] (::DataFrames.#kw##readtable)(::Array{Any,1}, ::DataFrames.#readtable, ::String) at .\<miss
 [4] include_string(::String, ::String) at .\loading.jl:522
 [5] include_string(::Module, ::String, ::String) at C:\Users\utilisateur\.julia\v0.6\Compat\src
 [6] execute_request(::ZMQ.Socket, ::IJulia.Msg) at C:\Users\utilisateur\.julia\v0.6\IJulia\src\
 [7] (::Compat.#inner#14{Array{Any,1},IJulia.#execute_request,Tuple{ZMQ.Socket,IJulia.Msg}})() a
 [8] eventloop(::ZMQ.Socket) at C:\Users\utilisateur\.julia\v0.6\IJulia\src\eventloop.jl:8
```

```
 [9] (::IJulia.##15#18)() at .\task.jl:335
while loading In[2], in expression starting on line 1
```

In [3]:
```
# Split into training, validation and test sets (50%/25%/25%)
n = length(y)
val_start = round(Int, 0.50 * n)
test_start = round(Int, 0.75 * n)
train_X = X[1:val_start - 1, :]
train_y = y[1:val_start - 1]
val_X = X[val_start:test_start - 1, :]
val_y = y[val_start:test_start - 1]
test_X = X[test_start:end, :]
test_y = y[test_start:end];
```

In [4]:
```
#See the size of training set and test set
println("Size of training set:",size(train_X),size(train_y))
println("Size of validation set:",size(val_X),size(val_y))
println("Size of test set:",size(test_X),size(test_y))
```

```
Size of training set:(252, 13)(252,)
Size of validation set:(127, 13)(127,)
Size of test set:(127, 13)(127,)
```

In [5]:
```
# write the training functions for different types of linear regressions

##### standard linear regression #####
function standardlinear(X, y)
    # OutputFlag=0 to hide output from solver
    m = Model(solver=GurobiSolver(OutputFlag=0))
    p = size(X, 2) #nb of columns

    #variables
    @variable(m, t)
    @variable(m, β[1:p])

    # Constraints
    @constraint(m, norm(y - X * β) <= t)

    # Objective
    @objective(m, Min, t)

    solve(m)

    return getvalue(β)
end

##### lasso linear regression #####
```

```
function lassolinear(X, y, ρ)
    # OutputFlag=0 to hide output from solver
    m = Model(solver=GurobiSolver(OutputFlag=0))
    p = size(X, 2) #nb of columns

    #variables
    @variable(m, t)
    @variable(m, β[1:p])
    @variable(m, a[1:p])

    # Constraints
    @constraint(m, norm(y - X * β) <= t)
    @constraint(m, -a[1:p] .<= β[1:p])
    @constraint(m, β[1:p] .<= a[1:p])
    @constraint(m, a[1:p] .>= 0)

    # Objective
    @objective(m, Min, t + ρ * sum(a[j] for j = 1:p))

    solve(m)
    return getvalue(β)
end

##### ridge linear regression #####
function ridgelinear(X, y, ρ)
    # OutputFlag=0 to hide output from solver
    m = Model(solver=GurobiSolver(OutputFlag=0))

    p = size(X, 2) #nb of columns

    # Variables
    @variable(m, t)
    @variable(m, u)
    @variable(m, β[1:p])

    # Constraints
    @constraint(m, norm(y - X * β) <= t)
    @constraint(m, norm(β) <= u)

    # Objective
    @objective(m, Min, t + ρ * u)

    solve(m)
    return getvalue(β)
end
```

Out[5]: ridgelinear (generic function with 1 method)

```
In [6]: function findBestRho(train_X,
```

```
                      train_y,
                      val_X,
                      val_y,
                      rho_list)
    p = size(train_X, 2)
    k = length(rho_list)
    #instantiate arrays
    β_lasso_list = zeros(k, p)
    β_ridge_list = zeros(k, p)
    lasso_scores = zeros(k)
    ridge_scores = zeros(k)

    for i in 1:length(rho_list)
        # training on train sets for both regression methods

        β_lasso_list[i, :] = lassolinear(train_X, train_y, rho_list[i])
        #println("\nβ_lasso for rho =", rho_list[i], β_lasso_list[i, :])
        β_ridge_list[i, :] = ridgelinear(train_X, train_y, rho_list[i])
        #println("\nβ_ridge for rho =", rho_list[i], β_ridge_list[i, :])

        # performance metrics on validation sets for both regression methods
        lasso_scores[i] = norm(val_y - val_X * β_lasso_list[i, :])
        ridge_scores[i] = norm(val_y - val_X * β_ridge_list[i, :])


    end
    #println(lasso_scores)
    #println(ridge_scores)
    argmin_lasso = indmin(lasso_scores)
    argmin_ridge = indmin(ridge_scores)

    return rho_list[argmin_lasso], rho_list[argmin_ridge]
end
```

Out[6]: findBestRho (generic function with 1 method)

In [7]:
```
rho_list = [0.001, 0.01, 0.1, 1, 2]

best_rho = findBestRho(train_X, train_y, val_X, val_y, rho_list)

println("Best rho for lasso: ", best_rho[1] )
println("Best rho for ridge: ", best_rho[2])
```

```
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Academic license - for non-commercial use only
```

```
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Best rho for lasso: 0.1
Best rho for ridge: 2.0
```

```
In [8]: # retrain the whole model with training and validation sets together

        new_train_X = vcat(train_X, val_X)
        new_train_y = vcat(train_y, val_y)

        # find best beta for standard linear regression
        β_standard = standardlinear(new_train_X, new_train_y)
        println("\nBest β for standard linear regression is: ", β_standard)

        # find best beta for lasso

        ρ_lasso = best_rho[1]
        β_lasso = lassolinear(new_train_X, new_train_y, ρ_lasso)
        println("\nBest β for lasso is: ", β_lasso)

        # find best beta for ridge

        ρ_ridge = best_rho[2]
        β_ridge = ridgelinear(new_train_X, new_train_y, ρ_ridge)
        println("\nBest β for ridge is: ", β_ridge)
```

```
Academic license - for non-commercial use only

Best β for standard linear regression is: [-0.182837, 0.0463545, 0.0566232, 0.814289, -5.54437,
Academic license - for non-commercial use only

Best β for lasso is: [-0.152615, 0.0472997, 0.0245548, 0.525413, -0.0077624, 6.19149, -0.0105729
Academic license - for non-commercial use only

Best β for ridge is: [-0.119194, 0.0540906, 0.0293727, 0.521321, -0.0163899, 5.33832, 0.00387411
```

```
In [9]: # score standard linear regression
        score_standard = norm(test_y - test_X * β_standard)
        println("Standard linear regression score: ", score_standard)

        # score lasso
        score_lasso = norm(test_y - test_X * β_lasso)
        println("Lasso score: ", score_lasso)
```

```
        # score ridge
        score_ridge = norm(test_y - test_X * β_ridge)
        println("Ridge score: ", score_ridge)

        # baseline
        train_y_mean = mean(new_train_y) #use mean on training and validation sets
        score_baseline = norm(test_y - train_y_mean)
        println("Baseline score: ", score_baseline)

        # compare scores of regression with the baseline model
        println("\nRelative gap standard linear regression % baseline: ", (score_baseline - scor
        println("Relative gap lasso % baseline: ", (score_baseline - score_lasso)*100/score_base
        println("Relative gap ridge % baseline: ", (score_baseline - score_ridge)*100/score_base
```

```
Standard linear regression score: 91.31777137966286
Lasso score: 89.47021659084704
Ridge score: 83.59529327926523
Baseline score: 129.2265350398843

Relative gap standard linear regression % baseline: 29.335123508899578 %
Relative gap lasso % baseline: 30.764825843830703 %
Relative gap ridge % baseline: 35.3110464089558 %
```

b Notebook 2: using communities-and-crimes.csv

```
In [1]: using JuMP, Gurobi, DataFrames
```

```
In [2]: df = readtable("communities-and-crime.csv",header=false)
        X = Matrix(df[1:end - 1])
        y = df[end];
```

```
WARNING: readtable is deprecated, use CSV.read from the CSV package instead
Stacktrace:
 [1] depwarn(::String, ::Symbol) at .\deprecated.jl:70
 [2] #readtable#232(::Bool, ::Char, ::Array{Char,1}, ::Char, ::Array{String,1}, ::Array{String,1
 [3] (::DataFrames.#kw##readtable)(::Array{Any,1}, ::DataFrames.#readtable, ::String) at .\<miss
 [4] include_string(::String, ::String) at .\loading.jl:522
 [5] include_string(::Module, ::String, ::String) at C:\Users\utilisateur\.julia\v0.6\Compat\src
 [6] execute_request(::ZMQ.Socket, ::IJulia.Msg) at C:\Users\utilisateur\.julia\v0.6\IJulia\src\
 [7] (::Compat.#inner#14{Array{Any,1},IJulia.#execute_request,Tuple{ZMQ.Socket,IJulia.Msg}})() a
 [8] eventloop(::ZMQ.Socket) at C:\Users\utilisateur\.julia\v0.6\IJulia\src\eventloop.jl:8
 [9] (::IJulia.##15#18)() at .\task.jl:335
while loading In[2], in expression starting on line 1
```

```
In [3]: # Split into training, validation and test sets (50%/25%/25%)
        n = length(y)
```

```
val_start = round(Int, 0.50 * n)
test_start = round(Int, 0.75 * n)
train_X = X[1:val_start - 1, :]
train_y = y[1:val_start - 1]
val_X = X[val_start:test_start - 1, :]
val_y = y[val_start:test_start - 1]
test_X = X[test_start:end, :]
test_y = y[test_start:end];
```

In [4]:
```
#See the size of training set and test set
println("Size of training set:",size(train_X),size(train_y))
println("Size of validation set:",size(val_X),size(val_y))
println("Size of test set:",size(test_X),size(test_y))
```

```
Size of training set:(60, 122)(60,)
Size of validation set:(31, 122)(31,)
Size of test set:(31, 122)(31,)
```

In [5]:
```
# write the training functions for different types of linear regressions

##### standard linear regression #####
function standardlinear(X, y)
    # OutputFlag=0 to hide output from solver
    m = Model(solver=GurobiSolver(OutputFlag=0))
    p = size(X, 2) #nb of columns

    #variables
    @variable(m, t)
    @variable(m, β[1:p])

    # Constraints
    @constraint(m, norm(y - X * β) <= t)

    # Objective
    @objective(m, Min, t)

    solve(m)
    return getvalue(β)
end

##### lasso linear regression #####
function lassolinear(X, y, ρ)
    # OutputFlag=0 to hide output from solver
    m = Model(solver=GurobiSolver(OutputFlag=0))

    p = size(X, 2) #nb of columns
```

```
    #variables
    @variable(m, t)
    @variable(m, β[1:p])
    @variable(m, a[1:p])

    # Constraints
    @constraint(m, norm(y - X * β) <= t)
    @constraint(m, -a[1:p] .<= β[1:p])
    @constraint(m, β[1:p] .<= a[1:p])
    @constraint(m, a[1:p] .>= 0)


    # Objective
    @objective(m, Min, t + ρ * sum(a[j] for j = 1:p))

    solve(m)

    return getvalue(β)
end

function lassolinear(X, y, ρ)
    # OutputFlag=0 to hide output from solver
    m = Model(solver=GurobiSolver(OutputFlag=0))

    p = size(X, 2) #nb of columns



    #variables
    @variable(m, t)
    @variable(m, β[1:p])
    @variable(m, a[1:p])

    # Constraints
    @constraint(m, norm(y - X * β) <= t)
    @constraint(m, -a[1:p] .<= β[1:p])
    @constraint(m, β[1:p] .<= a[1:p])
    @constraint(m, a[1:p] .>= 0)


    # Objective
    @objective(m, Min, t + ρ * sum(a[j] for j = 1:p))

    solve(m)

    return getvalue(β)
```

```
    end

    ##### ridge linear regression #####
    function ridgelinear(X, y, ρ)
        # OutputFlag=0 to hide output from solver
        m = Model(solver=GurobiSolver(OutputFlag=0))
        p = size(X, 2) #nb of columns

        # Variables
        @variable(m, t)
        @variable(m, u)
        @variable(m, β[1:p])

        # Constraints
        @constraint(m, norm(y - X * β) <= t)
        @constraint(m, norm(β) <= u)

        # Objective
        @objective(m, Min, t + ρ * u)

        solve(m)

        return getvalue(β)
    end
```

Out[5]: ridgelinear (generic function with 1 method)

In [6]:
```
function findBestRho(train_X,
                     train_y,
                     val_X,
                     val_y,
                     rho_list)
    p = size(train_X, 2)
    k = length(rho_list)
    #instantiate arrays
    β_lasso_list = zeros(k, p)
    β_ridge_list = zeros(k, p)
    lasso_scores = zeros(k)
    ridge_scores = zeros(k)

    for i in 1:length(rho_list)
        # training on train sets for both regression methods

        β_lasso_list[i, :] = lassolinear(train_X, train_y, rho_list[i])
        #println("\nβ_lasso for rho =", rho_list[i], β_lasso_list[i, :])
        β_ridge_list[i, :] = ridgelinear(train_X, train_y, rho_list[i])
        #println("\nβ_ridge for rho =", rho_list[i], β_ridge_list[i, :])
```

```
            # performance metrics on validation sets for both regression methods
            lasso_scores[i] = norm(val_y - val_X * β_lasso_list[i, :])
            ridge_scores[i] = norm(val_y - val_X * β_ridge_list[i, :])


        end
        #println(lasso_scores)
        #println(ridge_scores)
        argmin_lasso = indmin(lasso_scores)
        argmin_ridge = indmin(ridge_scores)

        return rho_list[argmin_lasso], rho_list[argmin_ridge]
    end
```

Out[6]: findBestRho (generic function with 1 method)

In [7]:
```
rho_list = [0.001, 0.01, 0.1, 1, 2]
best_rho = findBestRho(train_X, train_y, val_X, val_y, rho_list)
println("Best rho for lasso: ", best_rho[1] )
println("Best rho for ridge: ", best_rho[2])
```

```
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Best rho for lasso: 1.0
Best rho for ridge: 2.0
```

In [8]:
```
# retrain the whole model with training and validation sets together

new_train_X = vcat(train_X, val_X)
new_train_y = vcat(train_y, val_y)

# find best beta for standard linear regression
β_standard = standardlinear(new_train_X, new_train_y)
println("\nBest β for standard linear regression is: ", β_standard)

# find best beta for lasso

ρ_lasso = best_rho[1]
β_lasso = lassolinear(new_train_X, new_train_y, ρ_lasso)
println("\nBest β for lasso is: ", β_lasso)
```

```julia
    # find best beta for ridge

    ρ_ridge = best_rho[2]
    β_ridge = ridgelinear(new_train_X, new_train_y, ρ_ridge)
    println("\nBest β for ridge is: ", β_ridge)
```

```
Academic license - for non-commercial use only

Best β for standard linear regression is: [-0.0527188, 0.57872, -0.649981, -0.564125, -0.120323,
Academic license - for non-commercial use only

Best β for lasso is: [4.01425e-10, 1.11734e-9, 0.113693, -1.49524e-10, 5.02483e-10, 3.84554e-10,
Academic license - for non-commercial use only

Best β for ridge is: [-0.00179024, 0.0210995, 0.0604243, -0.0444612, 0.00730347, -0.0103841, 0.0
```

In [9]:

```julia
# score standard linear regression
score_standard = norm(test_y - test_X * β_standard)
println("Standard linear regression score: ", score_standard)

# score lasso
score_lasso = norm(test_y - test_X * β_lasso)
println("Lasso score: ", score_lasso)

# score ridge
score_ridge = norm(test_y - test_X * β_ridge)
println("Ridge score: ", score_ridge)

# baseline
train_y_mean = mean(new_train_y) #use mean on training and validation sets
score_baseline = norm(test_y - train_y_mean)
println("Baseline score: ", score_baseline)

# compare scores of regression with the baseline model
println("\nRelative gap standard linear regression % baseline: ", (score_baseline - scor
println("Relative gap lasso % baseline: ", (score_baseline - score_lasso)*100/score_base
println("Relative gap ridge % baseline: ", (score_baseline - score_ridge)*100/score_base
```

```
Standard linear regression score: 2.5479228082978107
Lasso score: 0.7833980234161126
Ridge score: 0.750864457615523
Baseline score: 1.4589372704759014

Relative gap standard linear regression % baseline: -74.64238249713678 %
Relative gap lasso % baseline: 46.30351562952598 %
Relative gap ridge % baseline: 48.53346522770084 %
```