Lab 2 - Methods in Linear Regression Problems Julia

February 6, 2024

1 Math 7243 Lab 2

Author: Jacob S. Zelko

Collaborators: Christopher Cesares, Philippos Dimitroglou

Date: February 6, 2024

Statement of Academic Integrity: Outside of course office hours, I collaborated with Chris Cesare and Phil Dimitroglou on this assignment.

1.1 Set-Up

Loading required packages:

[1]:
```julia
using Pkg
Pkg.activate("../.")
```

```
  Activating project at `~/School/machine_learning_1`
```

[2]:
```julia
using CairoMakie
using CSV
using DataFrames
using Distributions
using Downloads
using Images
using MLJ
using MLJLinearModels
using Statistics
using StatsBase
```

Download the AMES dataset:

[3]:
```julia
Downloads.download("https://raw.githubusercontent.com/tipthederiver/
  ↪Math-7243-2020/master/Datasets/Ames/train.csv", "ames.csv")
```

[3]:
```
"ames.csv"
```

1.1.1 Problem 1 Data Pre-Processing

Based on the Lab 2 master notebook, if the sum of the living area and basement square footage was less than 4000, we filter out the rows as follows:

[4]:
```
df = CSV.read("ames.csv", DataFrame)
filter!(row -> row.GrLivArea + row.BsmtUnfSF < 4000, df);
```

Additionally, per the master, we filter to only numeric columns of this dataframe:

[5]:
```
for c   names(df)
    if !(eltype(df[!, c]) <: Number)
        df[!, c] = tryparse.(Int, df[!, c])
    end
end
df = df[!, (<:).(eltype.(eachcol(df)), Union{Int, Missing})]
```

[5]:

	Id	MSSubClass	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	BsmtFinSF1
	Int64	Int64	Int64	Int64	Int64	Int64	Int64	Int64
1	1	60	8450	7	5	2003	2003	706
2	2	20	9600	6	8	1976	1976	978
3	3	60	11250	7	5	2001	2002	486
4	4	70	9550	7	5	1915	1970	216
5	5	60	14260	8	5	2000	2000	655
6	6	50	14115	5	5	1993	1995	732
7	7	20	10084	8	5	2004	2005	1369
8	8	60	10382	7	6	1973	1973	859
9	9	50	6120	7	5	1931	1950	0
10	10	190	7420	5	6	1939	1950	851
11	11	20	11200	5	5	1965	1965	906
12	12	60	11924	9	5	2005	2006	998
13	13	20	12968	5	6	1962	1962	737
14	14	20	10652	7	5	2006	2007	0
15	15	20	10920	6	5	1960	1960	733
16	16	45	6120	7	8	1929	2001	0
17	17	20	11241	6	7	1970	1970	578
18	18	90	10791	4	5	1967	1967	0
19	19	20	13695	5	5	2004	2004	646
20	20	20	7560	5	6	1958	1965	504
21	21	60	14215	8	5	2005	2006	0
22	22	45	7449	7	7	1930	1950	0
23	23	20	9742	8	5	2002	2002	0
24	24	120	4224	5	7	1976	1976	840
25	25	20	8246	5	8	1968	2001	188
26	26	20	14230	8	5	2007	2007	0
27	27	20	7200	5	7	1951	2000	234
28	28	20	11478	8	5	2007	2008	1218
29	29	20	16321	5	6	1957	1997	1277
30	30	30	6324	4	6	1927	1950	0
…	…	…	…	…	…	…	…	…

Finally, remove these columns per the template notebook:

[6]:
```
df = df[:, Not("MSSubClass")]
df = df[:, Not("Id")]
```
[6]:

	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	BsmtFinSF1	BsmtFinSF2	
	Int64	Int64	Int64	Int64	Int64	Int64	Int64	
1	8450	7	5	2003	2003	706	0	...
2	9600	6	8	1976	1976	978	0	...
3	11250	7	5	2001	2002	486	0	...
4	9550	7	5	1915	1970	216	0	...
5	14260	8	5	2000	2000	655	0	...
6	14115	5	5	1993	1995	732	0	...
7	10084	8	5	2004	2005	1369	0	...
8	10382	7	6	1973	1973	859	32	...
9	6120	7	5	1931	1950	0	0	...
10	7420	5	6	1939	1950	851	0	...
11	11200	5	5	1965	1965	906	0	...
12	11924	9	5	2005	2006	998	0	...
13	12968	5	6	1962	1962	737	0	...
14	10652	7	5	2006	2007	0	0	...
15	10920	6	5	1960	1960	733	0	...
16	6120	7	8	1929	2001	0	0	...
17	11241	6	7	1970	1970	578	0	...
18	10791	4	5	1967	1967	0	0	...
19	13695	5	5	2004	2004	646	0	...
20	7560	5	6	1958	1965	504	0	...
21	14215	8	5	2005	2006	0	0	...
22	7449	7	7	1930	1950	0	0	...
23	9742	8	5	2002	2002	0	0	...
24	4224	5	7	1976	1976	840	0	...
25	8246	5	8	1968	2001	188	668	...
26	14230	8	5	2007	2007	0	0	...
27	7200	5	7	1951	2000	234	486	...
28	11478	8	5	2007	2008	1218	0	...
29	16321	5	6	1957	1997	1277	0	...
30	6324	4	6	1927	1950	0	0	...
...	...	...	...	...	...	...	...	

1.1.2 Problem 2 Data Pre-Processing

Loading the MRI scan label information:

[7]:
```
labels = CSV.read("labels.csv", DataFrame)
```

[7]:

	Column1	Filename	ID	M/F	Hand	Age	Educ	SES
	Int64	String31	String15	String1	String1	Int64	Int64	Int64?
1	0	OAS1_0001_MR1_55.png	OAS1_0001_MR1	F	R	74	2	3
2	1	OAS1_0001_MR1_120.png	OAS1_0001_MR1	F	R	74	2	3
3	2	OAS1_0001_MR1_180.png	OAS1_0001_MR1	F	R	74	2	3
4	3	OAS1_0002_MR1_55.png	OAS1_0002_MR1	F	R	55	4	1
5	4	OAS1_0002_MR1_120.png	OAS1_0002_MR1	F	R	55	4	1
6	5	OAS1_0002_MR1_180.png	OAS1_0002_MR1	F	R	55	4	1
7	6	OAS1_0003_MR1_55.png	OAS1_0003_MR1	F	R	73	4	3
8	7	OAS1_0003_MR1_120.png	OAS1_0003_MR1	F	R	73	4	3
9	8	OAS1_0003_MR1_180.png	OAS1_0003_MR1	F	R	73	4	3
10	9	OAS1_0010_MR1_55.png	OAS1_0010_MR1	M	R	74	5	2
11	10	OAS1_0010_MR1_120.png	OAS1_0010_MR1	M	R	74	5	2
12	11	OAS1_0010_MR1_180.png	OAS1_0010_MR1	M	R	74	5	2
13	12	OAS1_0011_MR1_55.png	OAS1_0011_MR1	F	R	52	3	2
14	13	OAS1_0011_MR1_120.png	OAS1_0011_MR1	F	R	52	3	2
15	14	OAS1_0011_MR1_180.png	OAS1_0011_MR1	F	R	52	3	2
16	15	OAS1_0013_MR1_55.png	OAS1_0013_MR1	F	R	81	5	2
17	16	OAS1_0013_MR1_120.png	OAS1_0013_MR1	F	R	81	5	2
18	17	OAS1_0013_MR1_180.png	OAS1_0013_MR1	F	R	81	5	2
19	18	OAS1_0018_MR1_55.png	OAS1_0018_MR1	M	R	39	3	4
20	19	OAS1_0018_MR1_120.png	OAS1_0018_MR1	M	R	39	3	4
21	20	OAS1_0018_MR1_180.png	OAS1_0018_MR1	M	R	39	3	4
22	21	OAS1_0019_MR1_55.png	OAS1_0019_MR1	F	R	89	5	1
23	22	OAS1_0019_MR1_120.png	OAS1_0019_MR1	F	R	89	5	1
24	23	OAS1_0019_MR1_180.png	OAS1_0019_MR1	F	R	89	5	1
25	24	OAS1_0021_MR1_55.png	OAS1_0021_MR1	F	R	80	3	3
26	25	OAS1_0021_MR1_120.png	OAS1_0021_MR1	F	R	80	3	3
27	26	OAS1_0021_MR1_180.png	OAS1_0021_MR1	F	R	80	3	3
28	27	OAS1_0022_MR1_55.png	OAS1_0022_MR1	F	R	69	2	4
29	28	OAS1_0022_MR1_120.png	OAS1_0022_MR1	F	R	69	2	4
30	29	OAS1_0022_MR1_180.png	OAS1_0022_MR1	F	R	69	2	4
...	...	...	...	...	...	...	...	...

Per the instructions in this notebook, I downsampled each image by a downsampling rate of 8:

[8]:
```
# Downsampling rate
DS = 8

#=

Default image size (based on number of channels
per each image)

=#
im_size = 176 * 176

# Checking default sizes
```

```julia
if 30976/DS % 1 > 0
    DS = 1
    im_size = 30976
else
    im_size = Int(30976/DS)
end
```

[8]: 3872

Creating default matrix to hold downsampled image arrays:

[9]:
```julia
data = zeros(609, im_size)
```

[9]:
```
609×3872 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0

 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

Now, load each MRI image and downsample it; convert each downsampled image into a row vector for insertion into the data matrix:

[10]:
```julia
file_dir = "MRI_Images/"
for (i, file_name) in enumerate(labels.Filename)
    img = mean(Float64.(Gray.(Images.load(joinpath(file_dir, file_name)))), dims=3)[:, :, 1]
```

```
    img = vec(img)
    data[i, :] .= img[1:DS:end]
end
```

1.2 Problems

1.2.1 Problem 1: Bootstrapping a Confidence Interval

Description: Using a for loop, compute `beta_0` and `beta_1` 1000 times for samples of size $N = 1436$ with replacement and store their results in vectors. Additionally, generate and submit the following:

1. **Plot a histogram of `\beta_0` and `\beta_1`.**
2. **Sort the `beta_0` values and find the interval containing the middle 950 values. This is the bootstrap 95% confidence interval.**
3. **Compute the confidence interval. Remember that here you use all of the training data. Compare your results.**

Part 1 Load the linear regressor model:

[11]:
```
@load LinearRegressor pkg=MLJLinearModels verbosity=0
```

[11]:
```
LinearRegressor
```

Define my input data and target data

[12]:
```
X = DataFrame(:X => df[!, "1stFlrSF"])
y = df[!, :SalePrice]
```

[12]:
```
1436-element Vector{Int64}:
 208500
 181500
 223500
 140000
 250000
 143000
 307000
 200000
 129900
 118000
 129500
 345000
 144000

 112000
  92000
 136000
 287090
```

```
145000
 84500
185000
175000
210000
266500
142125
147500
```

Instantiate learning machine with a linear regression model

[13]:
```
mach = machine(LinearRegressor(), X, y)
```

```
  Warning: The number and/or types of data
arguments do not match what the specified model
  supports. Suppress this type check by specifying
`scitype_check_level=0`.

  Run `@doc MLJLinearModels.LinearRegressor` to learn more
about your model's requirements.

  Commonly, but non exclusively, supervised models are
constructed using the syntax
  `machine(model, X, y)` or `machine(model, X, y, w)` while
most other models are
  constructed with `machine(model, X)`.  Here `X` are
features, `y` a target, and `w`
  sample or class weights.

  In general, data in `machine(model, data…)` is expected
to satisfy

      scitype(data) <: MLJ.fit_data_scitype(model)

  In the present case:

  scitype(data) = Tuple{Table{AbstractVector{Count}},
AbstractVector{Count}}

  fit_data_scitype(model) =
Tuple{Table{<:AbstractVector{<:ScientificTypesBase.Continuous}},
AbstractVector{ScientificTypesBase.Continuous}}
  @ MLJBase

~/.julia/packages/MLJBase/mIaqI/src/machines.jl:231
```

[13]:
```
untrained Machine; caches model-specific representations of data
  model: LinearRegressor(fit_intercept = true, …)
  args:
```

```
1:   Source @698   Table{AbstractVector{Count}}
2:   Source @768   AbstractVector{Count}
```

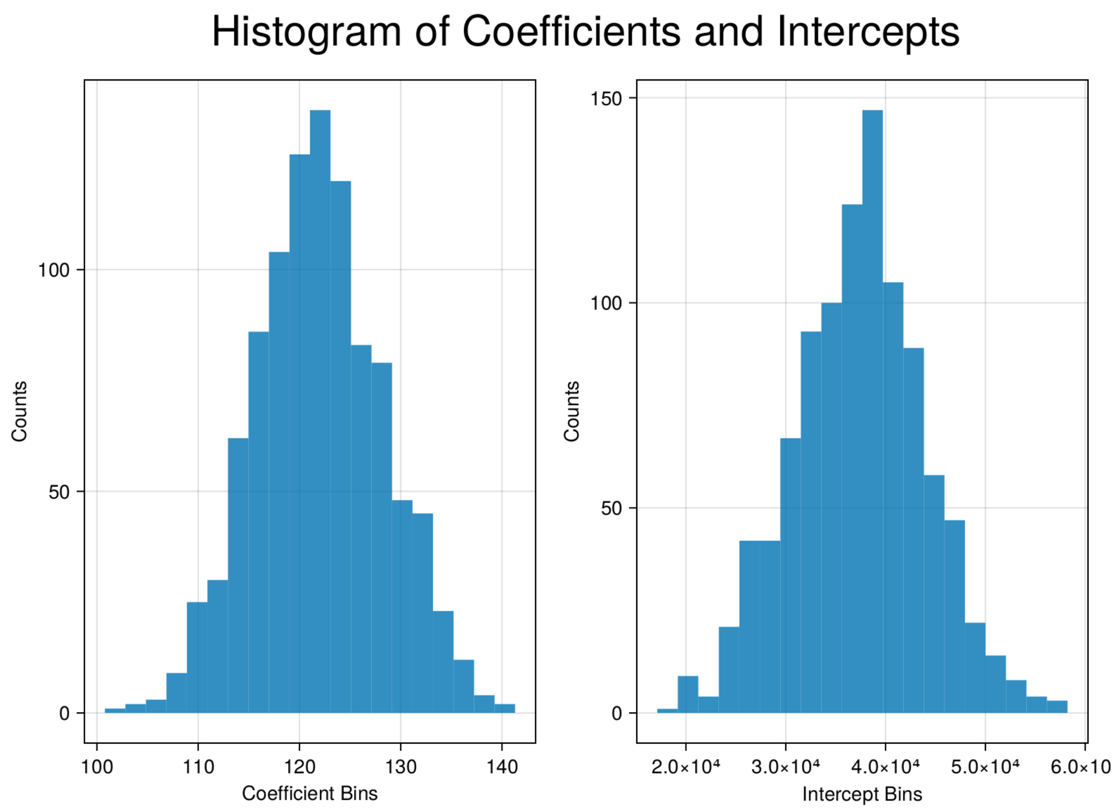
Generate intercept values (`beta_0`) and coefficient values (`beta_1`) and store them to a list:

[14]:
```
# How many iterations to run
N = 1000

# How much data for training?
training_proportion = .8 # 80/20 train/test split

# Number of samples for training
training_samples = round(Int, nrow(df) * .8)

beta_0 = []
beta_1 = []

for i in 1:N
  # Sample row indices used for training (with replacement)
  training_rows = sample(1:nrow(df), training_samples, replace=true)

  # Fit the learning machine to the training data
  fit!(mach, rows = training_rows, verbosity = 0)

  # Push values to list
  push!(beta_0, fitted_params(mach).intercept)
  push!(beta_1, fitted_params(mach).coefs[1][2])
end
```

Plot intercepts and coefficents:

[15]:
```
fig = Figure(; size = (800, 600));

ax1 = CairoMakie.Axis(fig[1, 1])
hist!(ax1, beta_1; bins = 20)
ax1.xlabel = "Coefficient Bins"
ax1.ylabel = "Counts"

ax2 = CairoMakie.Axis(fig[1, 2])
hist!(ax2, beta_0; bins = 20)
ax2.xlabel = "Intercept Bins"
ax2.ylabel = "Counts"

supertitle = Label(fig[0, :], "Histogram of Coefficients and Intercepts",␣
 ↪fontsize = 30)

fig
```

[15]:



Part 2 Sort the `beta_0` list

[16]:
```
sort_inds = sortperm(beta_0)
```

[16]:
```
1000-element Vector{Int64}:
 659
  26
 556
 280
 154
 385
 913
 977
 227
 314
 494
  27
 655

 925
```

```
 527
  99
 976
 665
 676
 852
 391
 293
 289
 276
 771
```

Get middle 950 values of `beta_0`

[17]:
```
sorted_beta_0 = beta_0[sort_inds]
sorted_beta_0[26:end-25]
```

[17]:
```
950-element Vector{Any}:
 24446.812833677945
 24544.16912225238
 24628.676815073806
 24761.635585108834
 24889.495273301472
 24965.007535435758
 25042.090874013495
 25169.589476493533
 25176.92969592272
 25206.967277758777
 25439.259412765903
 25469.9969262523
 25509.898758025836

 48920.34635719037
 48935.26414970009
 49105.03605251401
 49220.51801610896
 49414.97649089332
 49506.881642225475
 49729.500129890184
 49902.106808881035
 50029.06121614861
 50110.429004490674
 50268.93326605913
 50315.488889823
```

Part 3 Look up Z score at a 95% confidence interval.

[18]:
```
confidence_interval = 0.95

std_beta_0 = std(sorted_beta_0[26:end-25])
dn = Normal(0,1)
qt_beta_0 = quantile(dn, confidence_interval + (1 - confidence_interval)/2)
zscore_beta_0 = cdf(dn, qt_beta_0)
```

[18]:
```
0.9750000000000002
```

Calculate mean

[19]:
```
mean_beta_0 = mean(sorted_beta_0[26:end-25])
```

[19]:
```
37349.4874828346
```

Using these calculated values, we can compute the confidence intervals:

$$\left[\bar{x} - z_{\alpha/2}\left(\frac{\sigma}{\sqrt{n}}\right), \bar{x} + z_{\alpha/2}\left(\frac{\sigma}{\sqrt{n}}\right)\right]$$

As follows:

[20]:
```
[mean_beta_0 - zscore_beta_0 * (std_beta_0/sqrt(950)), mean_beta_0 +␣
↪zscore_beta_0 * (std_beta_0/sqrt(950))]
```

[20]:
```
2-element Vector{Float64}:
 37167.455763315294
 37531.519202353906
```

Comparing this to the histogram, the calculated interval makes sense as it contains the peak of the intercept value histogram which would contain 95 of the intercept values.

1.2.2 Problem 2: Linear Methods on High Dimensional Data

Description: Perform ridge regression and lasso regression on the MRI Slices dataset from canvas. Fit the MRI Slices data to the Normalized Whole-brain Volume (nWBV) in the labels data:

1. **Given a train-test split, what is the best α value for pure Ridge Regression? Justify your answer.**
2. **Given the train-test split, what is the best λ value for pure Lasso Regression? Justify your answer.**
3. **(Bonus) What is the best (α, λ) value for elastic net regression?**

Part 1 Prepare the data for prediction

[21]:
```
X = DataFrame(data, :auto)
y = labels.nWBV .|> Float64
```

[21]: 609-element Vector{Float64}:
```
 0.743
 0.743
 0.743
 0.81
 0.81
 0.81
 0.708
 0.708
 0.708
 0.689
 0.689
 0.689
 0.827

 0.748
 0.748
 0.748
 0.739
 0.739
 0.739
 0.818
 0.818
 0.818
 0.78
 0.78
 0.78
```

Load the ridge regressor model

[22]:
```
RidgeRegressor = @load RidgeRegressor pkg=MLJScikitLearnInterface
```

```
import MLJScikitLearnInterface
```

```
[ Info: For silent loading, specify
`verbosity=0`.
```

[22]:
```
MLJScikitLearnInterface.RidgeRegressor
```

Construct the learning maching built around the ridge regressor model:

[23]:
```
rr = RidgeRegressor()
mach = machine(rr, X, y)
```

[23]:
```
untrained Machine; caches model-specific representations of data
  model: RidgeRegressor(alpha = 1.0, …)
  args:
    1:  Source @122  Table{AbstractVector{ScientificTypesBase.Continuous}}
```

```
    2:  Source @044   AbstractVector{ScientificTypesBase.Continuous}
```

Create a training and test split based on an 80/20 train/test split:

[24]:
```
train, test = partition(eachindex(y), 0.8)
```

[24]:
```
([1, 2, 3, 4, 5, 6, 7, 8, 9, 10  …  478, 479, 480, 481, 482, 483, 484, 485, 486,
487], [488, 489, 490, 491, 492, 493, 494, 495, 496, 497  …  600, 601, 602, 603,
604, 605, 606, 607, 608, 609])
```

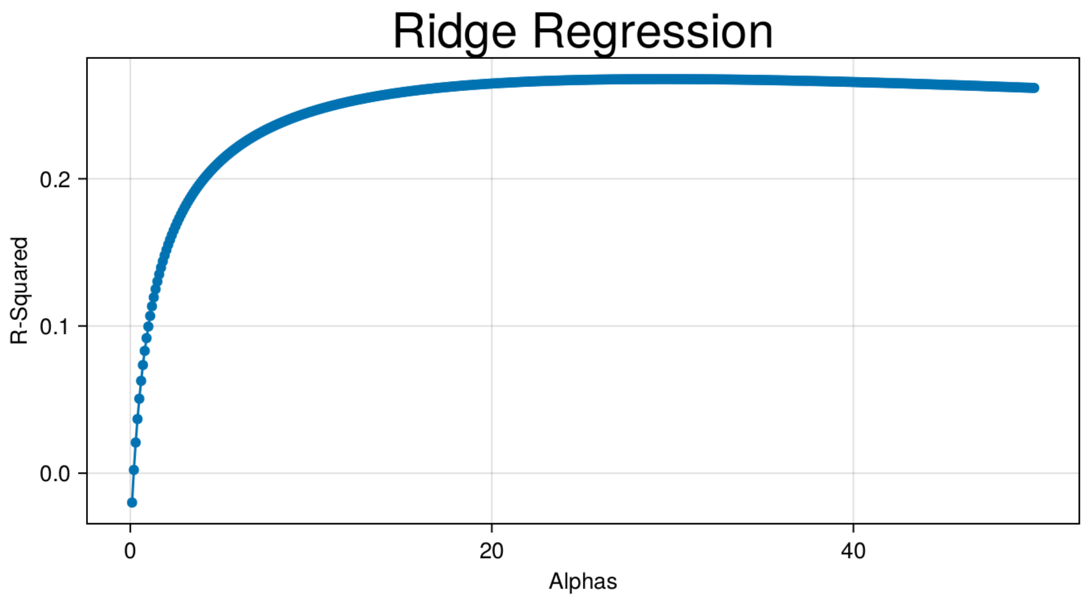
Evalute the machine upon changing the alpha value and calculate the model's R-Squared value:

[25]:
```
alphas = []
rsqs = []
for alp in .1:.1:50
    rr.alpha = alp
    output = MLJ.evaluate!(mach, resampling = [(train, test)], measure=[rsq])

    push!(alphas, alp)
    push!(rsqs, output.measurement)
end
```

Plot R-Squared value compared to each alpha:

[26]:
```
fig = Figure(; size = (700, 400));
ax = CairoMakie.Axis(fig[1, 1])
scatterlines!(ax, convert.(Float64, alphas), [Float64(x[1]) for x in rsqs])
ax.xlabel = "Alphas"
ax.ylabel = "R-Squared"

Label(fig[:, :, Top()], "Ridge Regression", fontsize = 30)

fig
```

[26]:



Based on the calculated R-Squared values, the value of alpha that maximizes R-Squared is:

[27]:
```
alphas[findmax(rsqs)[2]]
```

[27]:
```
29.5
```

Therefore, this alpha is the best for fitting the data. (Please see the addendum at the end of the notebook for discussion on this particular value.)

Part 2 Prepare the data for prediction

[28]:
```
X = DataFrame(data, :auto)
y = labels.nWBV .|> Float64
```

[28]:
```
609-element Vector{Float64}:
 0.743
 0.743
 0.743
 0.81
 0.81
 0.81
 0.708
 0.708
 0.708
 0.689
 0.689
 0.689
```

```
0.827

0.748
0.748
0.748
0.739
0.739
0.739
0.818
0.818
0.818
0.78
0.78
0.78
```

Load the lasso regressor model

[29]:
```
LassoRegressor = @load LassoRegressor pkg=MLJLinearModels
```

```
import MLJLinearModels
```

```
[ Info: For silent loading, specify
`verbosity=0`.
```

[29]:
```
MLJLinearModels.LassoRegressor
```

Construct the learning maching built around the lasso regressor model:

[30]:
```
lr = LassoRegressor(;)
mach = machine(lr, X, y)
```

[30]:
```
untrained Machine; caches model-specific representations of data
  model: LassoRegressor(lambda = 1.0, …)
  args:
    1:  Source @971   Table{AbstractVector{ScientificTypesBase.Continuous}}
    2:  Source @788   AbstractVector{ScientificTypesBase.Continuous}
```

Evalute the machine upon changing the lambda value and calculate the model's R-Squared value:

[31]:
```
lambdas = []
rsqs = []
for lam in 0.0001:.0001:0.001
    lr.lambda = lam
    output = MLJ.evaluate!(mach, resampling = [(train, test)], measure=[rsq])

    push!(lambdas, lam)
    push!(rsqs, output.measurement)
    println(lam)
end
```

```
Warning: Proximal GD did not converge in
1000 iterations.
 @ MLJLinearModels

~/.julia/packages/MLJLinearModels/yYgtO/src/fit/proxgrad.jl:73

0.0001
0.0002

 Warning: Proximal GD did not converge in
1000 iterations.
 @ MLJLinearModels

~/.julia/packages/MLJLinearModels/yYgtO/src/fit/proxgrad.jl:73

0.0003

 Warning: Proximal GD did not converge in
1000 iterations.
 @ MLJLinearModels

~/.julia/packages/MLJLinearModels/yYgtO/src/fit/proxgrad.jl:73

0.0004

 Warning: Proximal GD did not converge in
1000 iterations.
 @ MLJLinearModels

~/.julia/packages/MLJLinearModels/yYgtO/src/fit/proxgrad.jl:73

0.0005

 Warning: Proximal GD did not converge in
1000 iterations.
 @ MLJLinearModels

~/.julia/packages/MLJLinearModels/yYgtO/src/fit/proxgrad.jl:73

0.0006

 Warning: Proximal GD did not converge in
1000 iterations.
 @ MLJLinearModels

~/.julia/packages/MLJLinearModels/yYgtO/src/fit/proxgrad.jl:73

0.0007

 Warning: Proximal GD did not converge in
1000 iterations.
 @ MLJLinearModels

~/.julia/packages/MLJLinearModels/yYgtO/src/fit/proxgrad.jl:73

0.0008
```

```
 Warning: Proximal GD did not converge in
1000 iterations.
 @ MLJLinearModels

~/.julia/packages/MLJLinearModels/yYgtO/src/fit/proxgrad.jl:73
```

```
0.0009
```

```
 Warning: Proximal GD did not converge in
1000 iterations.
 @ MLJLinearModels

~/.julia/packages/MLJLinearModels/yYgtO/src/fit/proxgrad.jl:73
```

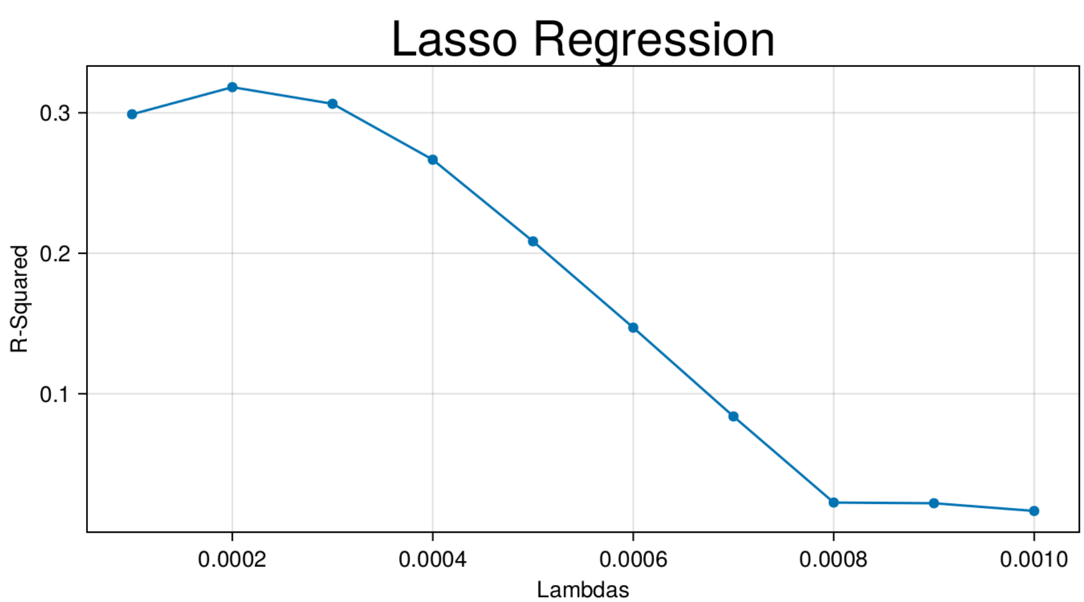
```
0.001
```

```
 Warning: Proximal GD did not converge in
1000 iterations.
 @ MLJLinearModels

~/.julia/packages/MLJLinearModels/yYgtO/src/fit/proxgrad.jl:73
```

Plot R-Squared value compared to each lambda:

[32]:
```
fig = Figure(; size = (700, 400));
ax = CairoMakie.Axis(fig[1, 1])
scatterlines!(ax, convert.(Float64, lambdas), [Float64(x[1]) for x in rsqs])
ax.xlabel = "Lambdas"
ax.ylabel = "R-Squared"

Label(fig[:, :, Top()], "Lasso Regression", fontsize = 30)

fig
```

[32]:



Based on the calculated R-Squared values, the value of lambda that maximizes R-Squared is:

[33]:
```
lambdas[findmax(rsqs)[2]]
```

[33]: 0.0002

Therefore, this lambda is the best for fitting the data. (Please see the addendum at the end of the notebook for discussion on this particular value.)

1.3 Addendum: Additional Justification Discussion for Problem 2

When I first reviewed my answers for problem 2, I originally thought they were incorrect as, when I collaborated with Chris and Phil, they got different answers. Even though I was using a different library and language, the math, I thought should still be the same. In many ways, that was correct as each instance had corresponding objective functions and cost functions that looked much like they did in lecture. That said, I wanted to justify further in particular why I think these values are reasonable according to the specific implementation I chose.

Within Python, there are two main libraries we have been taught to use – `statsmodels` and `scikit-learn`. The latter is particularly used for many ML applications. In the master notebook for Lab 2, it was suggested to use the `statsmodels` API to use ordinary least squares (OLS) as the curve fitting algorithm when given different linear regression models. Under the hood, instead, based on what was given in the master notebook, we use the `fit_regularized` function from the API to create a regularized fit to our linear regression model using an elastic net. The loss function that gets minimized in this process is:

$$\mathbf{Loss}(\beta) = \frac{1}{2n}\mathbf{RSS} + \alpha\left(\frac{(1-\lambda)}{2}\sum_i \beta_i^2 + \lambda\sum_i |\beta_i|\right)$$

Confusingly, depending on what parameters you pass to this model, you will get either a ridge or lasso regression which is what we are to model for this problem. I found this rather confusing and additionally looked at `scikit-learn`'s implementation which was similar. However, upon further exploration, I found that `scikit-learn` would actively suggest to use different models than an elastic net when fitting to a model. Some suggestions were to use their specific ridge or lasso regression models that were designed specifically to handle boundary conditions where the models performed poorly. So, I decided to do just that.

For my implementation, I used the `MLJ` package which had separate implementations of ridge and lasso regression. Additionally, `MLJ` offers an interface directly to `scikit-learn` that would yield the exact same implementations. To demonstrate the robustness of my approach, I used the `RidgeRegressor` from `scikit-learn` and the `LassoRegressor` from `MLJ`. I trained my learning machines on a 80/20% train/test split of the MRI data and associated labels and then had to explore choosing alphas and lambdas that best fit the data. Curiously, I only discovered this after reviewing the ridge and lasso regression notebook how drastically these values can change depending on the implementation.

The master notebook is where I received the interval to test for determining what the best alpha was for ridge regression and the ridge and lasso regression notebook was where I found the interval to test for the lambda. What I realized the difference came down to was that implementations really impact exactly what ends up working best for fitting a model. The `LassoRegressor`'s objective function:

$$Loss(\beta) = \frac{|X\theta - y|^2}{2} + \lambda|\theta|_1$$

is really still the elastic net function we are minimizing but just simplified to the RSS and $L1$ term without mixing occurring per what the problems were asking. So, in conclusion, I believe that although my answers may not "look" like other answers, my values are reasonable and my calculation of an R-Squared value should additionally justify my values in a manner somewhat agnostic to the exact implementation.