Homework 5

1. Fitting non-quadratic losses to data. In non_quadratic.json, you will find a 500 × 300 matrix U_train and a 500-vector v_train consisting of raw training input and output data, and a 500 × 300 matrix U_test and a 500-vector v_test consisting of raw test input and output data, respectively. We will work with input and output embeddings $x = \phi(u) = (1, u)$ and $y = \psi(v) = v$. Our performance metric is the RMS error on the test data set.

In regression_fit.jl we have also provided you with a function

\[
\text{regression_fit}(X, Y, l, r, \lambda).
\]

This function takes in input/output data $X$ and $Y$, a loss function $l(\hat{y}, y)$, a local regularizer function $r(\theta)$, and a local regularization hyper-parameter $\lambda$. It outputs the model parameters $\theta$ for the RERM linear predictor. You must include the Flux and LinearAlgebra Julia packages in your code in order to utilize this function. You will use this function to fit a linear predictor to the given data using the loss functions listed below.

- Quadratic loss: $\ell(\hat{y}, y) = (\hat{y} - y)^2$.
- Absolute loss: $\ell(\hat{y}, y) = |\hat{y} - y|$.
- Huber loss, with $\alpha \in \{0.5, 1, 2\}$: $\ell(\hat{y}, y) = p_\text{hub}^\alpha(\hat{y} - y)$, where
  \[
p_\text{hub}^\alpha(r) = \begin{cases} 
  r^2 & |r| \leq \alpha \\
  \alpha^2(2|r| - \alpha) & |r| > \alpha.
  \end{cases}
\]
- Log Huber loss, with $\alpha \in \{0.5, 1, 2\}$: $\ell(\hat{y}, y) = p_\text{dh}^\alpha(\hat{y} - y)$, where
  \[
p_\text{dh}^\alpha(y) = \begin{cases} 
  y^2 & |y| \leq \alpha \\
  \alpha^2(1 - 2\log(\alpha) + \log(y^2)) & |y| > \alpha.
  \end{cases}
\]

We won't use regularization so you can use $r(\theta) = 0$ and $\lambda = 1$ (though your choice of $\lambda$ does not matter).

Report the training and test RMS errors. Which model performs best? Create a one-sentence conjecture or story about why the particular model was the best one.

Julia hint. You will need to define the loss functions above in Julia. You can do this in a compact (but readable) form by defining the function inline, for example, for quadratic loss, $l_\text{quadratic}(\hat{y}, y) = (\hat{y} - y)^2$. You will also need to do the same for the regularizer (although it is zero); you can do this with $r(\theta) = 0$.

Solution.
<table>
<thead>
<tr>
<th>Method</th>
<th>Train RMS error</th>
<th>Test RMS error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square</td>
<td>0.734</td>
<td>1.39</td>
</tr>
<tr>
<td>Absolute</td>
<td>0.949</td>
<td>1.20</td>
</tr>
<tr>
<td>0.5-Huber</td>
<td>0.858</td>
<td>1.26</td>
</tr>
<tr>
<td>1-Huber</td>
<td>0.787</td>
<td>1.31</td>
</tr>
<tr>
<td>2-Huber</td>
<td>0.739</td>
<td>1.39</td>
</tr>
<tr>
<td>0.5-Log Huber</td>
<td>1.09</td>
<td>1.15</td>
</tr>
<tr>
<td>1-Log Huber</td>
<td>0.942</td>
<td>1.22</td>
</tr>
<tr>
<td>2-Log Huber</td>
<td>0.766</td>
<td>1.36</td>
</tr>
</tbody>
</table>

The log Huber predictor with $\alpha = 0.5$ performs the best on the test data. A reasonable guess as to why this did well is that there are many outliers in the data (although this is not the only conjecture that was reasonable).
In [1]:

using LinearAlgebra  
# using Statistics  
using Random  
using Flux  

include("readclassjson.jl")  
include("writeclassjson.jl")  
include("regression_fit.jl")  

import PyCall  
import PyPlot; const plt = PyPlot; plt.plt.style.use("seaborn")  

Random.seed!(0)

Out[1]: MersenneTwister(0)

In [ ]:

# n = 1000  
# d = 300  

# X = .1*randn(n,d)  
# y = [ 0.69*sum(X[i, 100:300]) - 0.42*sum(X[i, 1:100]) + 0.69 + 0.01*rand() for i=1:n]  

# outlier_idxs = shuffle(1:1000)[1:250]  
# y[outlier_idxs] = -y[outlier_idxs]

# global X_train = X[1:500, :)  
# X_test = X[501:1000, :)  

# global y_train = y[1:500]  
# y_test = y[501:1000]  

# # outlier_idxs_train = shuffle(1:500)[1:50]  
# # y_train[outlier_idxs_train] = -y_train[outlier_idxs_train]

# # outlier_idxs_test = shuffle(501:1000)[1:100]  
# # y_test[outlier_idxs_test] = -y_test[outlier_idxs_test]

# Data = Dict()  
# Data["U_train"] = X_train  
# Data["v_train"] = y_train  
# Data["U_test"] = X_test  
# Data["v_test"] = y_test  

# writeclassjson(Data, "non_quadratic.json")

In [ ]:

Data = readclassjson("non_quadratic.json")  
X_train = Data["U_train"]  
X_test = Data["U_test"]  
y_train = Data["v_train"]  
y_test = Data["v_test"]
In [3]: #---------------------------

    function huber(r, alpha)
        if abs(r) <= alpha
            return r.^2
        else
            return alpha .* (2 .* abs(r) - alpha)
        end
    end

    function log_huber(y, α)
        if abs(y) <= α
            return y.^2
        else
            return (α.^2)*(1 - 2*log(α) + log(y.^2))
        end
    end

    function RMSE(yhat, y)
        mse = sum((yhat .- y).^2)/length(y)
        return sqrt(mse)
    end

    function p_tlt(r, τ)
        if r < 0
            return -τ * r
        else
            return (1-τ) * r
        end
    end

    #---------------------------

    loss_huber(yhat, y) = huber(yhat - y, 1)
    loss_log_huber(yhat, y) = log_huber(yhat - y, 1)
    loss_huber_half(yhat, y) = huber(yhat - y, 0.5)
    loss_log_huber_half(yhat, y) = log_huber(yhat - y, 0.5)
    loss_huber2(yhat, y) = huber(yhat - y, 2)
    loss_log_huber2(yhat, y) = log_huber(yhat - y, 2)
    loss_square(yhat, y) = norm(yhat - y,2).^2
    # loss_tlt_25(yhat, y) = p_tlt(yhat - y[1], 0.25)
    # loss_tlt_75(yhat, y) = p_tlt(yhat - y[1], 0.75)
    loss_abs(yhat, y) = norm(yhat - y,1)

    #---------------------------

    function l1_reg(theta)
        return norm(theta[2:end], 1)
    end

    none_reg(x) = 0

Out[3]: none_reg (generic function with 1 method)
In [1]:
losses = [loss_square, loss_abs, loss_huber, loss_huber_half, loss_huber
2, loss_log_huber, loss_log_huber_half, loss_log_huber2]
names = ["square", "absolute", "huber", "huber_half", "huber2", "log_hub
er", "log_huber_half", "log_huber2"]
train_rmses = Dict()
test_rmses = Dict()
for (LOSS, NAME) in zip(losses, names):
    theta = regression_fit([ones(500) * X_train], y_train, LOSS, none_reg, 1; numiters=100)
    train_rmses[NAME] = RMSE([ones(500) * X_train] * theta, y_train)
    test_rmses[NAME] = RMSE([ones(500) * X_test] * theta, y_test)
end

In [5]:
train_rmses

Out[5]: Dict{Any, Any} with 8 entries:
    "huber2"         => 0.739177
    "absolute"       => 0.948894
    "huber_half"     => 0.857814
    "log_huber"      => 0.94288
    "log_huber_half" => 1.08746
    "square"         => 0.734838
    "log_huber2"     => 0.766111
    "huber"          => 0.786936

In [6]:
test_rmses

Out[6]: Dict{Any, Any} with 8 entries:
    "huber2"         => 1.38951
    "absolute"       => 1.19533
    "huber_half"     => 1.25655
    "log_huber"      => 1.21788
    "log_huber_half" => 1.15124
    "square"         => 1.3935
    "log_huber2"     => 1.35633
    "huber"          => 1.31375
2. Non quadratic regularizers.

Using the same data file `non_quadratic.json` and provided `regresion_fit` function, we will investigate the impact of different regularizers.

Using the **test RMS error**, select the best 2 loss functions from Problem 1. We will evaluate them with the following regularizers:

- **Quadratic or ridge regularization**: \( r(\theta) = \lambda \|\theta_{2:k}\|_2^2 \)
- **Lasso regularization**: \( r(\theta) = \lambda \|\theta_{2:k}\|_1 \)
- **No regularization** (you’ll use this as a baseline)

You are free to choose the weight \( \lambda \). A good starting choice is 0.1 but you are encouraged to experiment. Keep in mind different weights may work better for different functions.

(a) For the 2 best loss functions in Problem 1, and the regularizers listed above, report the training and test RMS errors. What is the best loss + regularizer combination? Don’t forget to try a few different values of \( \lambda \) (you only have to report the one you choose).

(b) Provide a (brief) comment on whether your results in this problem agree with your results from Problem 1.

(c) Some loss functions, such as quadratic loss, are convex. Others are not (such as log-Huber). Convex functions are advantageous because they can be reliably optimized. Is your best loss + regularizer convex? If not, what is the best convex loss + regularizer you found?

**Solution.**

The two best loss functions from Problem 1 should be 0.5-log-Huber, with a test RMS error of 1.15, and absolute, with a test RMS error of 1.20.

In this problem, results may vary depending on the value of \( \lambda \). One sample of results with \( \lambda = 0.05 \) is shown. It is OK to have different results as long as \( \lambda \) is reported.

<table>
<thead>
<tr>
<th>Loss Function</th>
<th>Train RMS error</th>
<th>Test RMS error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5-Log Huber, no regularizer</td>
<td>1.09</td>
<td>1.15</td>
</tr>
<tr>
<td>0.5-Log Huber, quadratic, ( \lambda = 0.05 )</td>
<td>1.16</td>
<td>1.15</td>
</tr>
<tr>
<td>0.5-Log Huber, lasso, ( \lambda = 0.05 )</td>
<td>1.21</td>
<td>1.16</td>
</tr>
<tr>
<td>Absolute, no regularizer</td>
<td>0.95</td>
<td>1.20</td>
</tr>
<tr>
<td>Absolute, quadratic, ( \lambda = 0.05 )</td>
<td>1.16</td>
<td>1.14</td>
</tr>
<tr>
<td>Absolute, lasso, ( \lambda = 0.05 )</td>
<td>1.21</td>
<td>1.16</td>
</tr>
</tbody>
</table>

(a) The best loss + regularizer we found with \( \lambda = 0.05 \) was absolute loss \( l(\hat{y}, y) = |\hat{y} - y| \) with a square regularizer \( \|\theta_{2:k}\|_2^2 \). Depending on the choice of \( \lambda \), it is also possible to have 0.5-log-Huber loss perform better (for example, \( \lambda = 0.1 \) produces this result).
(b) A reasonable answer is that in Problem 1 the good performance of log-Huber suggests outliers in the data. Adding the regularizer reduces the impact of outliers. Since the regularizer improves our performance, the results here agree with Problem 1.

It’s possible to have other answers.

(c) The best convex loss + regularizer we found was absolute loss with a square regularizer. It is OK to have a different answer as long as you don’t confuse nonconvex and convex functions.
In [1]:
using LinearAlgebra
using Statistics
using Random
using Flux
using Plots

include("readclassjson.jl")
include("writeclassjson.jl")

import PyCall
import PyPlot; const plt = PyPlot; plt.plt.style.use("seaborn")

Random.seed!(0)

Out[1]: MersenneTwister(0)

In [2]: include("regression_fit.jl")

Out[2]: regression_fit (generic function with 1 method)

In [ ]: Data = readclassjson("non_quadratic.json")
X_train = Data["U_train"]
X_test = Data["U_test"]
y_train = Data["v_train"]
y_test = Data["v_test"]
In [11]:
    
    function RMSE(yhat, y)
        mse = sum((yhat .- y).^2)/length(y)
        return sqrt(mse)
    end

    loss_abs(yhat, y) = norm(yhat - y, 2)

    function log_huber(y, α)
        if abs(y) <= α
            return y.^2
        else
            return (α.^2)*(1 - 2*log(α) + log(y.^2))
        end
    end

    loss_log_huber_half(yhat, y) = log_huber(yhat - y, 0.5)

    function square_reg(theta)
        return norm(theta[2:end], 2).^2
    end

    function l1_reg(theta)
        return norm(theta[2:end], 1)
    end

    none_reg(x) = 0

Out[11]: none_reg (generic function with 1 method)

In [ ]:
    
    regs = [square_reg, l1_reg, none_reg,]
    regnames = ["square", "absolute", "none"]
    losses = [loss_abs, loss_log_huber_half,]
    lossnames = ["absolute", "log huber half"]
    train_rmses = Dict()
    test_rmses = Dict()
    for (LOSS, LNAME) in zip(losses, lossnames)
        for (REG, RNAME) in zip(regs, regnames)
            theta = regression_fit([ones(500) X_train], y_train, LOSS, REG, 0.05; numiters=100)

            train_rmses[(LNAME, RNAME)] = RMSE([ones(500) X_train]*theta, y_train)
            test_rmses[(LNAME, RNAME)] = RMSE([ones(500) X_test]*theta, y_test)
        end
    end
In [13]: train_rmses

Out[13]: Dict{Any, Any} with 6 entries:
  ("log huber half", "none")    => 1.08746
  ("log huber half", "square") => 1.18888
  ("absolute", "square")      => 1.15934
  ("absolute", "absolute")    => 1.20882
  ("absolute", "none")        => 0.948894
  ("log huber half", "absolute") => 1.21118

In [14]: test_rmses

Out[14]: Dict{Any, Any} with 6 entries:
  ("log huber half", "none")    => 1.15124
  ("log huber half", "square") => 1.15172
  ("absolute", "square")      => 1.14082
  ("absolute", "absolute")    => 1.1633
  ("absolute", "none")        => 1.19533
  ("log huber half", "absolute") => 1.16372
3. **How often does your predictor over-estimate?** In this problem, you will identify how often linear predictors with tilted absolute losses over-estimate.

In `residual_props.json`, you will find a $500 \times 10$ matrix $U_{\text{train}}$ and a 500-vector $v_{\text{train}}$ consisting of raw training input and output data, and a $500 \times 10$ matrix $U_{\text{test}}$ and a 500-vector $v_{\text{test}}$ consisting of raw test input and output data, respectively. We will work with input and output embeddings $x = \phi(u) = (1, u)$ and $y = \psi(v) = v$, and use no regularization ($r(\theta) = 0$). You will also use `regression_fit.jl` from the previous problem.

Recall that the tilted absolute penalty is

$$p_\tau(u) = \begin{cases} -\tau u & u < 0 \\ (1 - \tau)u & u \geq 0 \end{cases}$$

where $\tau \in [0,1]$. Fit a linear predictor to the given data using the tilted absolute penalty, i.e., $\ell(\hat{y}, y) = p_\tau(\hat{y} - y)$, for $\tau \in \{0.15, 0.5, 0.85\}$. For both the training set and the test set, report how frequently this predictor over-estimates, and plot the empirical CDFs of the residuals.

**Hint.** A predictor $\hat{y}$ over-estimates $y$ when $\hat{y} > y$. To generate an empirical CDF plot of the residuals $r$ of length $d$, you may plot $\text{collect}(1:d)/d$ versus $\text{sort}(r)$.

**Solution.**

For $\tau = 0.15$, the predictor overestimates 15% of the time on the training set and 14.8% on the test set. For $\tau = 0.5$, the predictor overestimates 50% of the time on the training set and 52.8% on the test set. For $\tau = 0.85$, the predictor overestimates 85% of the time on the training set and 81.8% on the test set.
In [1]:
```
using LinearAlgebra
using Statistics
using Random
using Flux

include("readclassjson.jl")
include("writeclassjson.jl")

import PyCall
import PyPlot; const plt = PyPlot; plt.plt.style.use("seaborn")

Random.seed!(0)
```

Out[1]: MersenneTwister(0)

In [2]:
```
include("regression_fit.jl")
```

Out[2]: regression_fit (generic function with 1 method)

In [ ]:
```
# n = 1000
# d = 10

# X = randn(n,d)
# y = [ X[i,3] + X[i,1] + X[i,7] - 0.7*X[i,6] + 0.2*X[i,9] - 2*(X[i,2].^2) + 0.69 + 0.1*randn() for i=1:n]

# # outlier_idxs = shuffle(1:1000)[1:100]
# # y[outlier_idxs] = -y[outlier_idxs]

# global X_train = X[1:500, :]
# X_test = X[501:1000, :]

# global y_train = y[1:500]
# y_test = y[501:1000]

# outlier_idxs_train = shuffle(1:500)[1:50]
# y_train[outlier_idxs_train] = -y_train[outlier_idxs_train]

# outlier_idxs_test = shuffle(501:1000)[1:50]
# y_test[outlier_idxs_test] = -y_test[outlier_idxs_test]

# Data = Dict()
# Data["U_train"] = X_train
# Data["v_train"] = y_train
# Data["U_test"] = X_test
# Data["v_test"] = y_test
# writeclassjson(Data, "residual_props.json")
```
In [1]:
```
Data = readclassjson("residual_props.json")
X_train = Data["U_train"]
X_test = Data["U_test"]
y_train = Data["v_train"]
y_test = Data["v_test"]
```

In [4]:
```
#-------------------------------------------------------

function RMSE(yhat, y)
    mse = sum((yhat .- y).^2)/length(y)
    return sqrt(mse)
end

function p_tlt(r, τ)
    if r < 0
        return -τ * r
    else
        return (1-τ) * r
    end
end

#-------------------------------------------------------

loss_tlt_15(yhat, y) = p_tlt(yhat - y, 0.15)
loss_tlt_50(yhat, y) = p_tlt(yhat - y, 0.5)
loss_tlt_85(yhat, y) = p_tlt(yhat - y, 0.85)

#-------------------------------------------------------

none_reg(x) = 0

#-------------------------------------------------------
```

In [5]:
```
losses = [loss_tlt_15, loss_tlt_50, loss_tlt_85]
names = ["tlt_15", "tlt_50", "tlt_85"]
train_rmses = Dict()
test_rmses = Dict()
train_res = Dict()
test_res = Dict()
for (LOSS, NAME) in zip(losses, names)
    theta = regression_fit([ones(500) X_train], y_train, LOSS, none_reg, 1; numiters=1000)
    train_rmses[NAME] = RMSE([ones(500) X_train]*theta, y_train)
    test_rmses[NAME] = RMSE([ones(500) X_test]*theta, y_test)
    train_res[NAME] = [ones(500) X_train]*theta - y_train
    test_res[NAME] = [ones(500) X_test]*theta - y_test
end
```

In [6]:
```
mean(train_res["tlt_15"] .> 0), mean(test_res["tlt_15"] .> 0)
```

Out[6]:
```
(0.15, 0.148)
```
In [7]: mean(train_res["tlt_50"] .> 0), mean(test_res["tlt_50"] .> 0)
Out[7]: (0.504, 0.528)

In [8]: mean(train_res["tlt_85"] .> 0), mean(test_res["tlt_85"] .> 0)
Out[8]: (0.856, 0.818)

In [19]: fig, ax = plt.subplots(2, 1, figsize=(7, 5))
   
   ax[1].hist(train_res["tlt_15"], bins=100)
   ax[1].set_title("0.15-tilted residuals, train")
   
   ax[2].hist(test_res["tlt_15"], bins=100)
   ax[2].set_title("0.15-tilted residuals, test")
   
   plt.tight_layout()
   plt.savefig("tlt15_plot.pdf", bbox_inches="tight")
In [10]: ```python
###train cumulative_fraction
fig, ax = plt.subplots(2,1)
ax[1].plot(cumsum(sort(train_res["tlt_15"])))
```

Out[10]: 1-element Vector{PyCall.PyObject}:
PyObject <matplotlib.lines.Line2D object at 0x7f4daa1abc10>

In [ ]: ```python
sort(train_res["tlt_15"])
```
In [12]: a, b = plt.hist(train_res["tlt_50"], bins=1000)
Out[12]: ([1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 ... 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [-1.4106035884096806, -1.39200070598722, -1.2617805011527568, -1.2431776152456413 ... 17.19228231870558], PyCall.PyObject[PyObject <matplotlib.patches.Rectangle object at 0x7f4db7f52cd0>, PyObject <matplotlib.patches.Rectangle object at 0x7f4db7f52f50>, PyObject <matplotlib.patches.Rectangle object at 0x7f4db7f52f10>, PyObject <matplotlib.patches.Rectangle object at 0x7f4db7f52c90>, ... PyObject <matplotlib.patches.Rectangle object at 0x7f4f1aad6e90>, PyObject <matplotlib.patches.Rectangle object at 0x7f4f1aae2a10>, PyObject <matplotlib.patches.Rectangle object at 0x7f4f1aae2dd0>], PyObject <matplotlib.patches.Rectangle object at 0x7f4f1aae2ed0>])
In [13]: fig, ax = plt.subplots(2,1)
for res in [train_res["tlt_15"], train_res["tlt_50"], train_res["tlt_85"]]
    a,b = plt.hist(res, bins=500)
    ax[1].plot(sort(res), cumsum(a)/maximum(cumsum(a)))
end
ax[1].set_xlim(-5,5)

Out[13]: (-5.0, 5.0)
In [14]: function cdfs(M, taus, name)
    
    trainresiduals = Any[]
    testresiduals = Any[]
    for i=1:3
        t = taus[i]
        setloss(M, TiltedLoss(t/10))
        train(M)
        push!(trainresiduals, predict_v_from_train(M) - Vtrain(M))
        push!(testresiduals, predict_v_from_test(M) - Vtest(M))
    end
    d = length(trainresiduals[1])
    y = collect(1:d)/d
    ax=plot(sort(trainresiduals[1][:]), y, linewidth=2, markersize=0, label="tau=$(taus[1]/10)"

    plot(ax, sort(trainresiduals[2][:]), y, linewidth=2, markersize=0, label="tau=$(taus[2]/10)"

    plot(ax, sort(trainresiduals[3][:]), y, linewidth=2, markersize=0, xlabel="training residual", ylabel="cumulative fraction", label="tau=$(taus[3]/10)"

                yticks=collect(0:0.1:1.0), legend="default", ylim=[-0.05,1.05], name="train_"*name)"

    d = length(testresiduals[1])
    y = collect(1:d)/d
    ax=plot(sort(testresiduals[1][:]), y, linewidth=2, markersize=0, label="tau=$(taus[1]/10)"

    plot(ax, sort(testresiduals[2][:]), y, linewidth=2, markersize=0, label="tau=$(taus[2]/10)"

    plot(ax, sort(testresiduals[3][:]), y, linewidth=2, markersize=0, label="tau=$(taus[3]/10)"

                xlabel="test residual", ylabel="cumulative fraction", legend="default", #ylim=[0,1],

            yticks=collect(0:0.1:1.0),name="test_"*name)"
    end

Out[14]: cdfs (generic function with 1 method)
In [15]:
res = train_res["tlt_15"]

d = length(res)
y = collect(1:d)/d

# ax=plt.plot(sort(res), y, linewidth=2, markersize=0, 
#       xlabel="training residual", ylabel="cumulative fraction", label="tau=", 
#       yticks=collect(0:0.1:1.0), legend="default", ylim=[-0.05,1.05])

fig, ax = plt.subplots(2,1)
d = length(res)
y = collect(1:d)/d
taus = [0.15, 0.50, 0.85]

ax[1].set_title("Train")
for (i,resname) in enumerate(["tlt_15", "tlt_50", "tlt_85"])
    res = train_res[resname]
    ax[1].plot(sort(res), y, label=string("τ=" , taus[i]))
end
ax[1].legend()

ax[2].set_title("Test")
for (i,resname) in enumerate(["tlt_15", "tlt_50", "tlt_85")
    res = test_res[resname]
    ax[2].plot(sort(res), y, label=string("τ=" , taus[i]))
end
ax[2].legend() 
plt.savefig("residual_cdfs.pdf", bbox_inches="tight")
Figure 1 CDFs for Problem 3.