Assignment 1

Gang Peng # 1002961921

February 23, 2020

The goal of this assignment is to get you familiar with the basics of decision theory and gradient-based model fitting.

1 Decision theory [13pts]

One successful use of probabilistic models is for building spam filters, which take in an email and take different actions depending on the likelihood that it's spam.

Imagine you are running an email service. You have a well-calibrated spam classifier that tells you the probability that a particular email is spam: $p(\text{spam}|\text{email})$. You have three options for what to do with each email: You can show it to the user, put it in the spam folder, or delete it entirely.

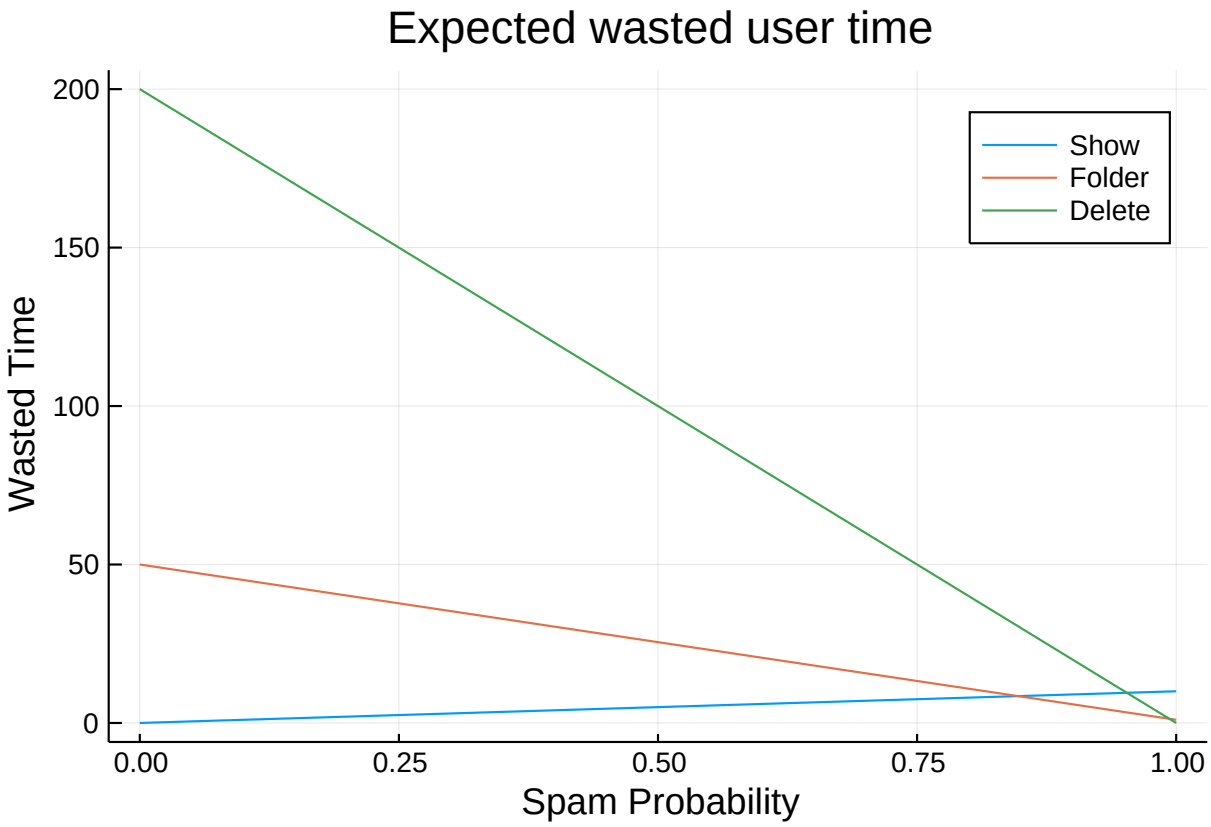
Depending on whether or not the email really is spam, the user will suffer a different amount of wasted time for the different actions we can take, $L(\text{action}, \text{spam})$:

Action	Spam	Not spam
Show	10	0
Folder	1	50
Delete	0	200

1. [3pts] Plot the expected wasted user time for each of the three possible actions, as a function of the probability of spam: $p(\text{spam}|\text{email})$

```
losses = [[10, 0],
          [1, 50],
          [0, 200]]


num_actions = length(losses)

function expected_loss_of_action(prob_spam, action)
    #TODO: Return expected loss over a Bernoulli random variable
    #      with mean prob_spam.
    #      Losses are given by the table above.

    expected = prob_spam .* losses[action][1] .+ (1 .- prob_spam) * losses[action][2]

    return expected
```

```
end

prob_range = range(0., stop=1., length=500)

# Make plot
using Plots
labels=["Show" "Folder" "Delete"]

p1=plot(prob_range, expected_loss_of_action(prob_range, 1),title="Expected wasted user
time",
         label=labels[1],xlabel="Spam Probability",ylabel="Wasted Time")
plot!(prob_range,expected_loss_of_action(prob_range, 2),label=labels[2])
display(plot!(prob_range,expected_loss_of_action(prob_range, 3),label=labels[3]))
```



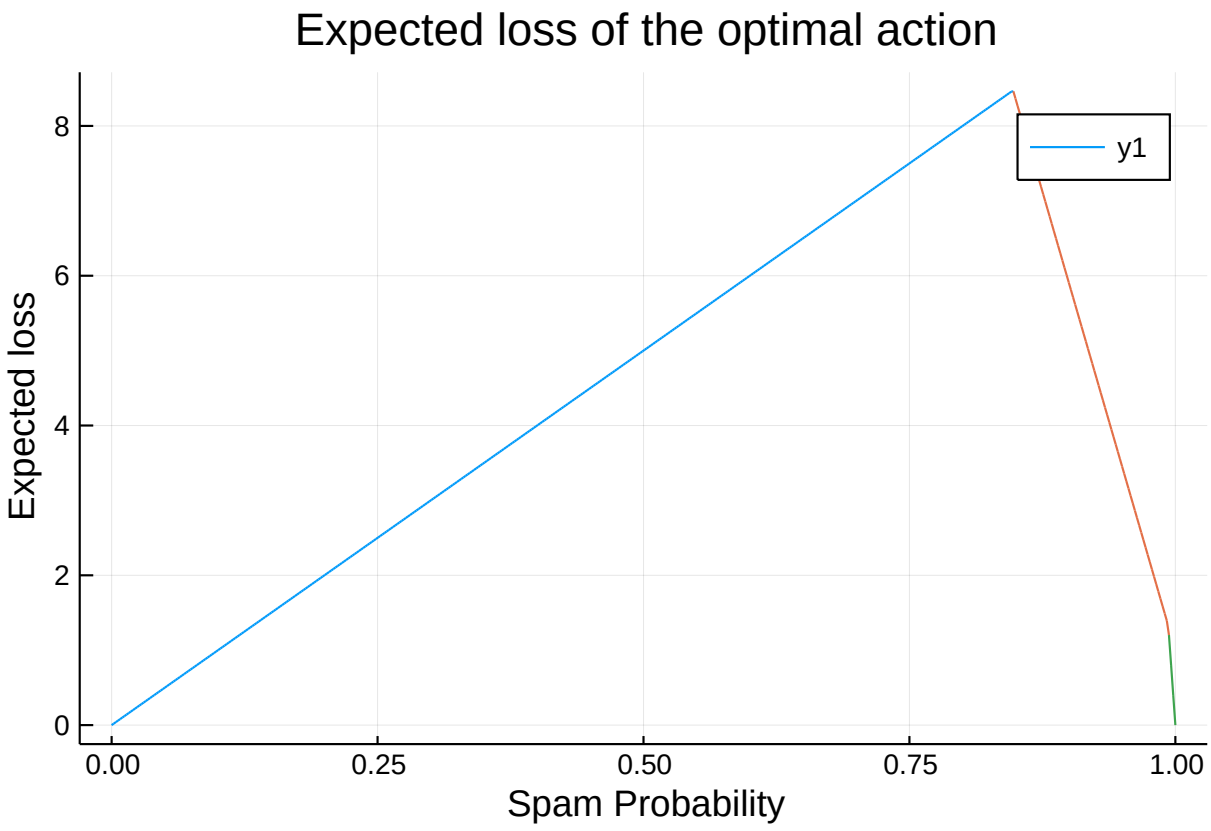
2. [2pts] Write a function that computes the optimal action given the probability of spam.

```
function optimal_action(prob_spam)
  #TODO: return best action given the probability of spam.
  # Hint: Julia's findmin function might be helpful.
  expected_losses=Array{Float64,1}(undef, num_actions)
  for action in 1:num_actions
      expected_losses[action]=expected_loss_of_action(prob_spam,action)
  end
  return findmin(expected_losses)
end

optimal_action (generic function with 1 method)
```

3. [4pts] Plot the expected loss of the optimal action as a function of the probability of spam.

Color the line according to the optimal action for that probability of spam.

```
prob_range = range(0, stop=1., length=500)
optimal_losses = []
optimal_actions = []
for p in prob_range
    # TODO:  Compute the optimal action and its expected loss for
    # probability of spam given by p.
    append!(optimal_losses, optimal_action(p)[1])
    append!(optimal_actions, optimal_action(p)[2])
end

plot(prob_range, optimal_losses, linecolor=optimal_actions,title="Expected loss of the
optimal action",
xlabel="Spam Probability",ylabel="Expected loss")
```



4. [4pts] For exactly which range of the probabilities of an email being spam should we delete an email?

Find the exact answer by hand using algebra.

We should delete the email when the expected_losses is smaller for delete than for folder or show.

That is the probobility of spam x satisfy:

```
x*1 + (1-x)*50 > (1-x)*200 and
x*10 > (1-x)*200
```

When $x > \frac{150}{151} \approx 0.9934$, we should delete the email.

2 Regression

2.1 Manually Derived Linear Regression [10pts]

Suppose that $X \in \mathbb{R}^{m \times n}$ with $n \geq m$ and $Y \in \mathbb{R}^n$, and that $Y \sim \mathcal{N}(X^T\beta, \sigma^2 I)$.

In this question you will derive the result that the maximum likelihood estimate $\hat{\beta}$ of β is given by

$$\hat{\beta} = (XX^T)^{-1}XY$$

1. [1pts] What happens if $n < m$?

This means number of observations smaller than number independent variables. $(XX^T)^{-1}$ does not exist. Then we will not have an estimation of β.

2. [2pts] What are the expectation and covariance matrix of $\hat{\beta}$, for a given true value of β?

Since $\hat{\beta} = (XX^T)^{-1}XY$, and $Y = X^T\beta + \epsilon$,

$$\begin{aligned}
\hat{\beta} &= (XX^T)^{-1}X(X^T\beta + \epsilon) \\
&= \beta + (XX^T)^{-1}X\epsilon \\
E[\hat{\beta}] &= E[\beta] + E[(XX^T)^{-1}X\epsilon] \\
&= \beta + (XX^T)^{-1}XE[\epsilon] \\
&= \beta
\end{aligned}$$

As we usually assume $E[\epsilon]$=0.

For the covariance matrix, since $\hat{\beta} - \beta = (XX^T)^{-1}X\epsilon$, we have

$$\begin{aligned}
E[(\hat{\beta} - \beta)(\hat{\beta} - \beta)^T] &= E[((XX^T)^{-1}X\epsilon)((XX^T)^{-1}X\epsilon)^T] \\
&= E[(XX^T)^{-1}X\epsilon\epsilon^T((XX^T)^{-1}X)^T] \\
&= E[\epsilon\epsilon^T](XX^T)^{-1}X((XX^T)^{-1}X)^T \\
&= \sigma^2 I \\
&= \sigma^2
\end{aligned}$$

If we assume X is non-stochastic.

3. [2pts] Show that maximizing the likelihood is equivalent to minimizing the squared error $\sum_{i=1}^n (y_i - x_i\beta)^2$. [Hint: Use $\sum_{i=1}^n a_i^2 = a^T a$]

The log-likelihood is:

$$
\begin{aligned}
l(\theta) &:= log(p(D|\theta)) \\
&= log(\prod_{i=1}^{N} p(y_i|X_i, \theta)) \\
&= \sum_{i=1}^{N} log(p(y_i|X_i, \theta))
\end{aligned}
$$

Usually use negative log-likelihood(NLL), and normal distribution to expand:

$$
\begin{aligned}
NLL(\theta) &= -\sum_{i=1}^{N} log(p(y_i|X_i, \theta)) \\
&= -\sum_{i=1}^{N} log[(\frac{1}{2\pi\sigma^2})^{\frac{1}{2}} exp(-\frac{1}{2\sigma^2}(y_i - X_i\beta)^2)] \\
&= -\sum_{i=1}^{N} (\frac{1}{2} log(\frac{1}{2\pi\sigma^2}) - \frac{1}{2\sigma^2}(y_i - X_i\beta)^2) \\
&= -\frac{N}{2} log(\frac{1}{2\pi\sigma^2}) - \frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - X_i\beta)^2 \\
&= -\frac{N}{2} log(\frac{1}{2\pi\sigma^2}) - \frac{1}{2\sigma^2} RSS(\beta)
\end{aligned}
$$

Since maximize likelihood is equivalent to minimize NLL, so it is equivalent to minimizing the squared error $RSS = \sum_{i=1}^{n} (y_i - x_i\beta)^2$.

4. [2pts] Write the squared error in vector notation, (see above hint), expand the expression, and collect like terms. [Hint: Use $\beta^T x^T y = y^T x\beta$ and $x^T x$ is symmetric]

The squared error in vector notation is,

$$
\begin{aligned}
\sum_{i=1}^{n} (y_i - x_i\beta)^2 &= (Y - X^T\beta)^T (Y - X^T\beta) \\
&= Y^T Y - \beta^T XY - YX^T\beta + \beta^T XX^T\beta \\
&= Y^T Y - 2\beta^T XY + \beta^T XX^T\beta
\end{aligned}
$$

5. [3pts] Use the likelihood expression to write the negative log-likelihood. Write the derivative of the negative log-likelihood with respect to β, set equal to zero, and solve to show the maximum likelihood estimate $\hat{\beta}$ as above.

As derived in question 3,

$$
\begin{aligned}
NLL(\theta) &= -\sum_{i=1}^{N} log(p(y_i|X_i, \theta)) \\
&= -\frac{N}{2} log(\frac{1}{2\pi\sigma^2}) - \frac{1}{2\sigma^2} RSS(\beta)
\end{aligned}
$$

To minimize NNL, is to minimize:

$$\begin{aligned} RSS(\beta) &= (Y - X^T\beta)^T(Y - X^T\beta) \\ &= Y^TY - 2\beta^TXY + \beta^TXX^T\beta \end{aligned}$$

Take derivative w.r.t β is

$$\frac{\partial RSS}{\partial \beta} = -2XY + 2(XX^T)\beta$$

Set the derivative to zero,

$$\begin{aligned} -2XY + 2(XX^T)\beta &= 0 \\ \hat{\beta} &= (XX^T)^{-1}XY \end{aligned}$$

2.2 Toy Data [2pts]

For visualization purposes and to minimize computational resources we will work with 1-dimensional toy data.

That is $X \in \mathbb{R}^{m \times n}$ where $m = 1$.

We will learn models for 3 target functions

- `target_f1`, linear trend with constant noise.
- `target_f2`, linear trend with heteroskedastic noise.
- `target_f3`, non-linear trend with heteroskedastic noise.

```
using LinearAlgebra

function target_f1(x, σ_true=0.3)
  noise = randn(size(x))
  y = 2x .+ σ_true.*noise
  return vec(y)
end

function target_f2(x)
  noise = randn(size(x))
  y = 2x + norm.(x)*0.3.*noise
  return vec(y)
end

function target_f3(x)
  noise = randn(size(x))
  y = 2x + 5sin.(0.5*x) + norm.(x)*0.3.*noise
  return vec(y)
end
```

```
target_f3 (generic function with 1 method)
```

1. [1pts] Write a function which produces a batch of data $x \sim \text{Uniform}(0, 20)$ and `y = target_f(x)`

```
using Random
using Distributions
function sample_batch(target_f, batch_size)
  x = rand(Uniform(0,20),1,batch_size)
  y = target_f(x)
  return (x,y)
end
```

```
sample_batch (generic function with 1 method)
```

```
using Test
@testset "sample dimensions are correct" begin
  m = 1 # dimensionality
  n = 200 # batch-size
  for target_f in (target_f1, target_f2, target_f3)
    x,y = sample_batch(target_f,n)
    @test size(x) == (m,n)
    @test size(y) == (n,)
  end
end
```

```
Test Summary:                  | Pass  Total
sample dimensions are correct |    6      6
Test.DefaultTestSet("sample dimensions are correct", Any[], 6, false)
```

2. [1pts] For all three targets, plot a $n = 1000$ sample of the data. **Note: You will use these plots later, in your writeup display once other questions are complete.**

```
using Plots
s1=sample_batch(target_f1,1000)
x1=s1[1]
y1=s1[2]
plot_f1 = plot(x1',y1,seriestype=:scatter,title="target_f1 Plot",xlabel="x1"
,ylabel="y1")

s2=sample_batch(target_f2,1000)
x2=s2[1]
y2=s2[2]
plot_f2 = plot(x2',y2,seriestype=:scatter,title="target_f2 Plot",xlabel="x2"
,ylabel="y2")

s3=sample_batch(target_f3,1000)
x3=s3[1]
y3=s3[2]
plot_f3 = plot(x3',y3,seriestype=:scatter,title="target_f3 Plot",xlabel="x3"
,ylabel="y3");
```

2.3 Linear Regression Model with $\hat{\beta}$ MLE [4pts]

1. [2pts] Program the function that computes the the maximum likelihood estimate given X and Y. Use it to compute the estimate $\hat{\beta}$ for a $n = 1000$ sample from each target function.

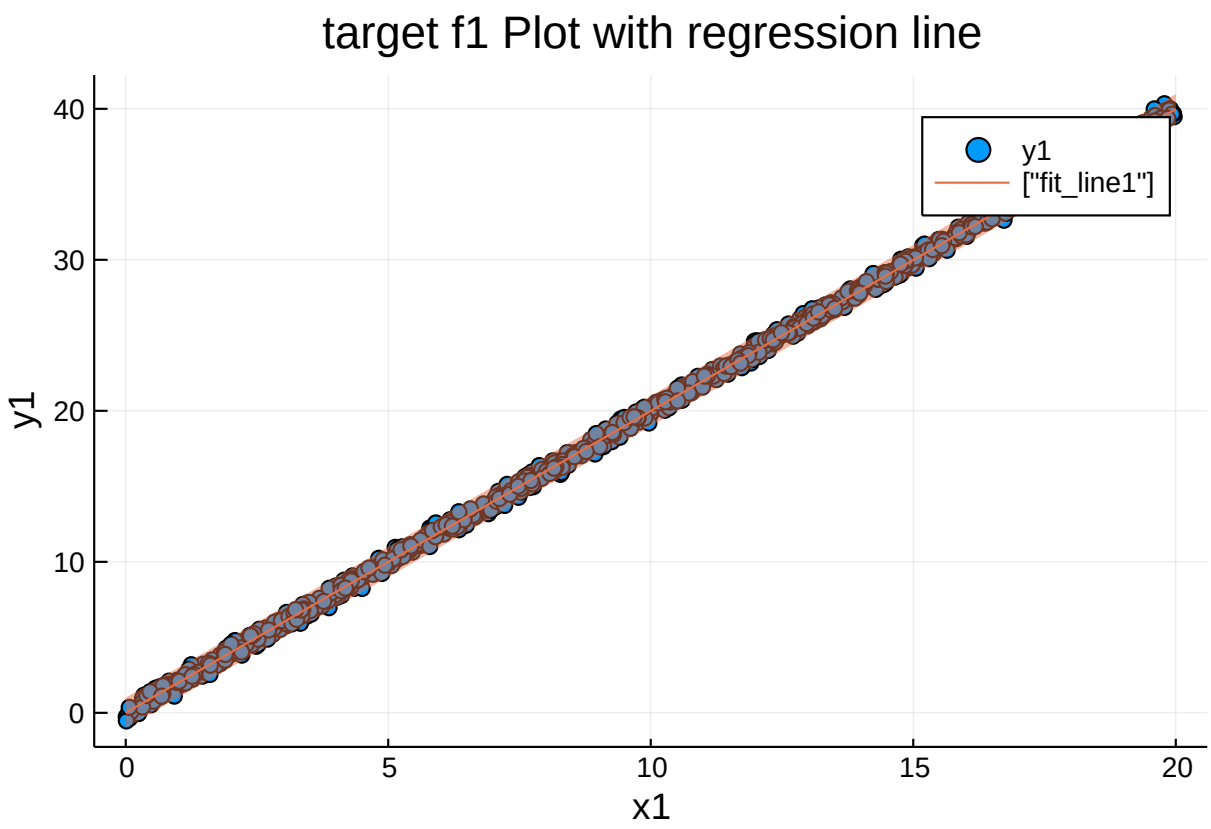
Since in the question we assume m=1, the formular for regression is: $Y_i = \beta_1 x_i + \beta_0$ then every β parameter is a array of two elments, denoting slope and intercept of the regression line.

```
using LinearAlgebra
function beta_mle(X,Y)
  beta = inv(X*X')*X*Y
  return beta
end

n=1000 # batch_size

#Use the same sample data that used to create plot_f1 above
x1=s1[1]
y1=s1[2]
x0=ones(1,n) #add a column of "1"s to multiply with β_0, the intercept of regression
line
X1=vcat(x0,x1)
β_mle_1 = beta_mle(X1,y1)

#Use the same sample data that used to create plot_f2 above
x2=s2[1]
y2=s2[2]
x0=ones(1,n)
X2=vcat(x0,x2)
β_mle_2 = beta_mle(X2,y2)

#Use the same sample data that used to create plot_f3 above
x3=s3[1]
y3=s3[2]
x0=ones(1,n)
X3=vcat(x0,x3)
β_mle_3 = beta_mle(X3,y3)
```

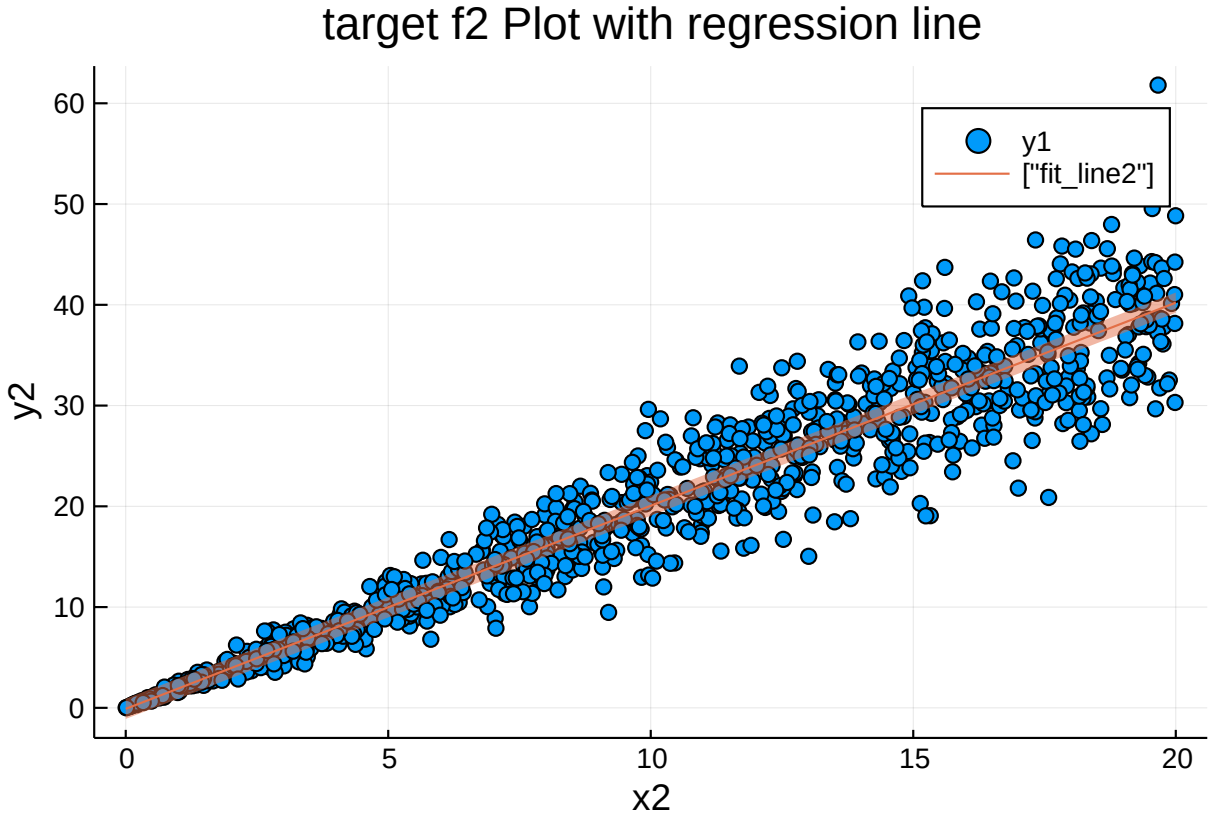
```
2-element Array{Float64,1}:
 1.3783394495450265
 1.9837584209653025
```

2. [2pts] For each function, plot the linear regression model given by $Y \sim \mathcal{N}(X^T\hat{\beta}, \sigma^2 I)$ for $\sigma = 1.$. This plot should have the line of best fit given by the maximum likelihood estimate, as well as a shaded region around the line corresponding to plus/minus one standard deviation (i.e. the fixed uncertainty $\sigma = 1.0$). Using `Plots.jl` this shaded uncertainty region can be achieved with the `ribbon` keyword argument. **Display 3 plots, one for each target function, showing samples of data and maximum likelihood estimate linear regression model**

```
x_range = range(0., stop=20., length=1000)
fit_line1=β_mle_1[2] .*x_range .+ β_mle_1[1]
plot!(plot_f1,x_range,fit_line1,ribbon=1.0,label=["fit_line1"],title="target f1 Plot with
regression line",xlabel="x1"
,ylabel="y1")
```

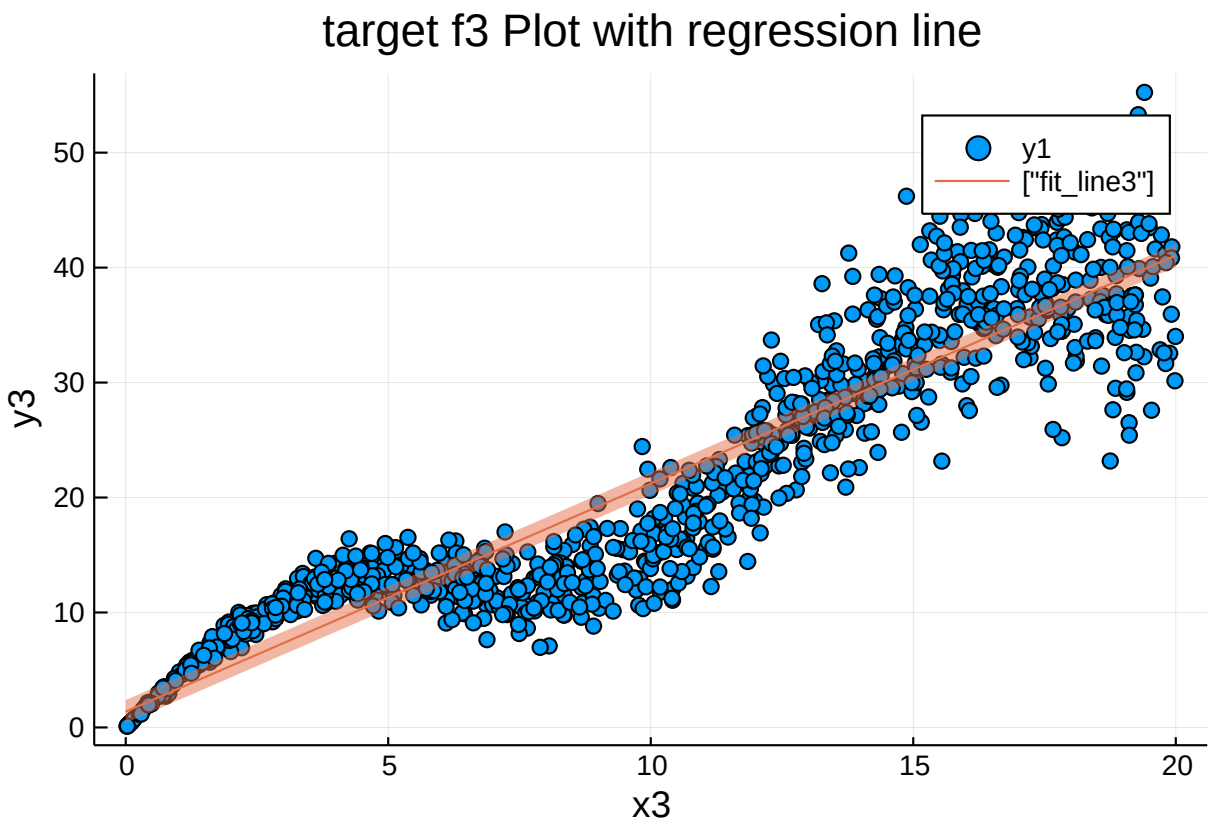



```
fit_line2=β_mle_2[2] .*x_range .+ β_mle_2[1]
plot!(plot_f2,x_range,fit_line2,ribbon=1,label=["fit_line2"],title="target f2 Plot with
regression line",xlabel="x2"
,ylabel="y2")
```




```
fit_line3=β_mle_3[2] .*x_range .+ β_mle_3[1]
```

```
plot!(plot_f3,x_range,fit_line3,ribbon=1.0,label=["fit_line3"],title="target f3 Plot with
regression line",xlabel="x3"
,ylabel="y3")
```



2.4 Log-likelihood of Data Under Model [6pts]

1. [2pts] Write code for the function that computes the likelihood of x under the Gaussian distribution $\mathcal{N}(\mu, \sigma)$. For reasons that will be clear later, this function should be able to broadcast to the case where x, μ, σ are all vector valued and return a vector of likelihoods with equivalent length, i.e., $x_i \sim \mathcal{N}(\mu_i, \sigma_i)$.

```
function gaussian_log_likelihood(μ, σ, x)
  """
  compute log-likelihood of x under N(μ,σ)
  """
  ll= log.(1 ./(sqrt(2*pi) .*σ) .*exp.(-1 ./(2 .*σ.^2) .*(x .- μ).^2))
  return ll
end
```

```
gaussian_log_likelihood (generic function with 1 method)
```

```
# Test Gaussian likelihood against standard implementation
@testset "Gaussian log likelihood" begin
using Distributions: pdf, Normal
# Scalar mean and variance
x = randn()
μ = randn()
σ = rand()
```

```
@test size(gaussian_log_likelihood(μ,σ,x)) == () # Scalar log-likelihood
@test gaussian_log_likelihood.(μ,σ,x) ≈ log.(pdf.(Normal(μ,σ),x)) # Correct Value

# Vector valued x under constant mean and variance
x = randn(100)
μ = randn()
σ = rand()
@test size(gaussian_log_likelihood.(μ,σ,x)) == (100,) # Vector of log-likelihoods
@test gaussian_log_likelihood.(μ,σ,x) ≈ log.(pdf.(Normal(μ,σ),x)) # Correct Values

# Vector valued x under vector valued mean and variance
x = randn(10)
μ = randn(10)
σ = rand(10)
@test size(gaussian_log_likelihood.(μ,σ,x)) == (10,) # Vector of log-likelihoods
@test gaussian_log_likelihood.(μ,σ,x) ≈ log.(pdf.(Normal.(μ,σ),x)) # Correct Values
end
```

```
Test Summary:           | Pass  Total
Gaussian log likelihood |    6      6
Test.DefaultTestSet("Gaussian log likelihood", Any[], 6, false)
```

2. [2pts] Use your gaussian log-likelihood function to write the code which computes the negative log-likelihood of the target value Y under the model $Y \sim \mathcal{N}(X^T\beta, \sigma^2 * I)$ for a given value of β.

If use the previous gaussian log-likelihhod function, the negative log-likelihood function is written as below,

```
function lr_model_nll(β,x,y;σ=1.)
  NLL= -sum(gaussian_log_likelihood(x'*β,σ,x))
end
```

```
lr_model_nll (generic function with 1 method)
```

But function as written above is very fragile, in fact if x =15, then gaussian likelihhod would be 0, and log-likelihood -inf, which is big problem in latter questions. Use formular derived in 2.1.3, can avoid this problem, and much more robust with random initial parameters inputs:

```
function lr_model_nll(β,x,y;σ=1.)
  NLL = -size(x)[2]/2 * log(1/(2*pi*σ)) .- 1/(2*σ^2) * (y' * y .- 2 * β'*x*y
.+β'*x*x'*β)
end
```

```
lr_model_nll (generic function with 1 method)
```

3. [1pts] Use this function to compute and report the negative-log-likelihood of a $n \in \{10, 100, 1000\}$ batch of data under the model with the maximum-likelihood estimate $\hat{\beta}$ and $\sigma \in \{0.1, 0.3, 1., 2.\}$ for each target function.

```
for n in (10,100,1000)
    println("--------  $n  ------------")
    for target_f in (target_f1,target_f2, target_f3)
      println("--------  $target_f  ------------")
      for σ_model in (0.1,0.3,1.,2.)
```

```
        println("--------  $σ_model  ------------")
        x,y = sample_batch(target_f,n)
        β_mle = beta_mle(x,y)
        nll = lr_model_nll(β_mle,x,y;σ=σ_model)
        println("Negative Log-Likelihood: $nll")
      end
    end
end
```

```
--------  10  ------------
--------  target_f1  ------------
--------  0.1  ------------
Negative Log-Likelihood: -31.9515129077932
--------  0.3  ------------
Negative Log-Likelihood: -0.35735339182324743
--------  1.0  ------------
Negative Log-Likelihood: 8.47230448591435
--------  2.0  ------------
Negative Log-Likelihood: 12.605004365680792
--------  target_f2  ------------
--------  0.1  ------------
Negative Log-Likelihood: -4312.447713624127
--------  0.3  ------------
Negative Log-Likelihood: -391.62555571532755
--------  1.0  ------------
Negative Log-Likelihood: -8.81593564048731
--------  2.0  ------------
Negative Log-Likelihood: -6.534022621817691
--------  target_f3  ------------
--------  0.1  ------------
Negative Log-Likelihood: -10075.706597212209
--------  0.3  ------------
Negative Log-Likelihood: -943.9761397436049
--------  1.0  ------------
Negative Log-Likelihood: -64.85934513104658
--------  2.0  ------------
Negative Log-Likelihood: -22.84241624719548
--------  100  ------------
--------  target_f1  ------------
--------  0.1  ------------
Negative Log-Likelihood: -449.1133350478192
--------  0.3  ------------
Negative Log-Likelihood: -18.406462315411048
--------  1.0  ------------
Negative Log-Likelihood: 86.78818950213459
--------  2.0  ------------
Negative Log-Likelihood: 125.53964885707362
--------  target_f2  ------------
--------  0.1  ------------
Negative Log-Likelihood: -61007.74044488674
--------  0.3  ------------
Negative Log-Likelihood: -6166.114758408097
--------  1.0  ------------
Negative Log-Likelihood: -492.4204397160757
--------  2.0  ------------
Negative Log-Likelihood: -46.74041350037035
--------  target_f3  ------------
--------  0.1  ------------
Negative Log-Likelihood: -111495.09636310751
--------  0.3  ------------
```

```
Negative Log-Likelihood: -16615.96094980363
--------  1.0  ------------
Negative Log-Likelihood: -709.8154907932803
--------  2.0  ------------
Negative Log-Likelihood: -153.24342607336123
--------  1000  ------------
--------  target_f1  ------------
--------  0.1  ------------
Negative Log-Likelihood: -4545.736789420477
--------  0.3  ------------
Negative Log-Likelihood: -162.71010624750164
--------  1.0  ------------
Negative Log-Likelihood: 873.3889589058886
--------  2.0  ------------
Negative Log-Likelihood: 1254.4333712259838
--------  target_f2  ------------
--------  0.1  ------------
Negative Log-Likelihood: -636927.754401866
--------  0.3  ------------
Negative Log-Likelihood: -67191.78930869365
--------  1.0  ------------
Negative Log-Likelihood: -5439.355500166763
--------  2.0  ------------
Negative Log-Likelihood: -183.36027759915964
--------  target_f3  ------------
--------  0.1  ------------
Negative Log-Likelihood: -1.1639399952837287e6
--------  0.3  ------------
Negative Log-Likelihood: -129194.63929844284
--------  1.0  ------------
Negative Log-Likelihood: -10409.857442193541
--------  2.0  ------------
Negative Log-Likelihood: -1549.0031390202778
```

4. [1pts] For each target function, what is the best choice of σ?

From the printed info, we can see in most cases, NLL numebrs when σ=0.1 are not stable, for example there is a huge drop of NLL for target f3 when σ=0.1 changed to σ=0.3 and n=1000. As expected too small values of σ have precision issues. Especially for target f3, which has bigger variance, small σ choice will not reflect the true variance of the distribution.

To choose the right σ, maybe better to look at data with bigger batch size.

For target f3, σ=0.1 data not precise, σ=0.3 and σ=1 NLL number are close, σ=2 NLL too large. So choose σ=1 for better precision.

For target f2, situation more or less similar to target_f3, also choose σ=1.

For target f1, the ture distribution known to be have small variance, and this reflected in the result, choose σ=1 or 0.3 not very different. We can choose either one.

2.5 Automatic Differentiation and Maximizing Likelihood [3pts]

In a previous question you derived the expression for the derivative of the negative log-likelihood with respect to β. We will use that to test the gradients produced by automatic differentiation.

1. [3pts] For a random value of β, σ, and $n = 100$ sample from a target function, use automatic differentiation to compute the derivative of the negative log-likelihood of the sampled data with respect β. Test that this is equivalent to the hand-derived value.

This is what I am talking about in the section of negative log-likelihood function, if lr*model*nll() function used gaussian log-likelihood function, then in most cases, the test will fail, because gaussian likelihood extreamly small and evaluated to zero in Julia, then lr*model*nll() -inf and ad_grad NaN.

```
function lr_model_nll(β,x,y;σ=1.)
  NLL= -sum(gaussian_log_likelihood(x'*β,σ,x))
end

β_test=0.5
σ_test=0
x=ones(1,100)
y=15 .*ones(100)
lr_model_nll(β_test,x,y;σ=σ_test)[1]
```

```
NaN
```

Using the derived formular for the lr*model*nll(), then test easily passsed:

```
function lr_model_nll(β,x,y;σ=1.)
  NLL = -size(x)[2]/2 * log(1/(2*pi*σ)) .- 1/(2*σ^2) * (y' * y .- 2 * β'*x*y
.+β'*x*x'*β)
end

using Zygote: gradient

@testset "Gradients wrt parameter" begin
β_test = randn()
σ_test = rand()
x,y = sample_batch(target_f1,100)
f(β)=lr_model_nll(β,x,y;σ=σ_test)[1]
ad_grad = gradient(f,β_test)
hand_derivative =  -1/(2*σ_test^2) .*(-2 .*x*y .+2 .*(x*x')*β_test)[1]
@test ad_grad[1] ≈ hand_derivative
end
```

```
Test Summary:           | Pass  Total
Gradients wrt parameter |    1      1
Test.DefaultTestSet("Gradients wrt parameter", Any[], 1, false)
```

2.5.1 Train Linear Regression Model with Gradient Descent [5pts]

In this question we will compute gradients of of negative log-likelihood with respect to β. We will use gradient descent to find β that maximizes the likelihood.

1. [3pts] Write a function `train_lin_reg` that accepts a target function and an initial estimate for β and some hyperparameters for batch-size, model variance, learning rate, and number of iterations. Then, for each iteration:
   - sample data from the target function
   - compute gradients of negative log-likelihood with respect to β

– update the estimate of β with gradient descent with specified learning rate

and, after all iterations, returns the final estimate of β.

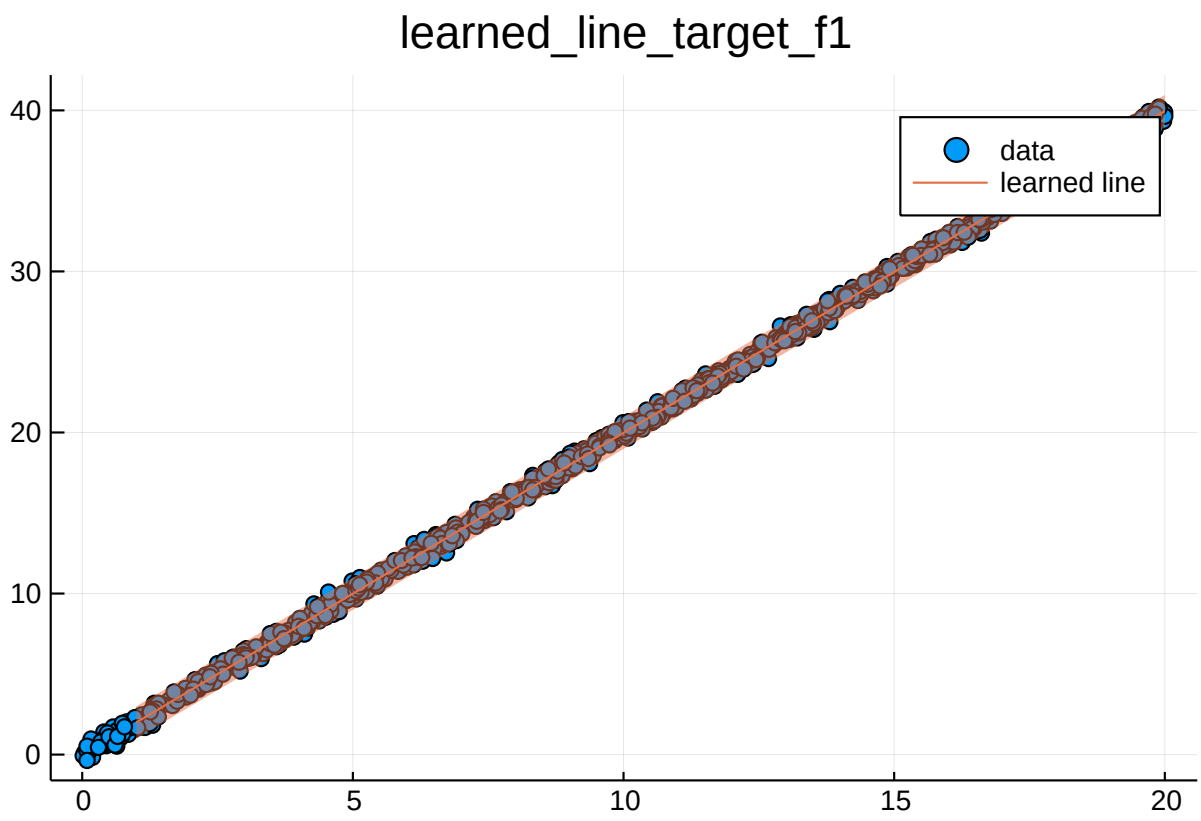
```
using Logging # Print training progress to REPL, not pdf

function train_lin_reg(target_f, β_init; bs= 100, lr = 1e-6, iters=1000, σ_model = 1. )
    β_curr = β_init
    for i in 1:iters
      x,y = sample_batch(target_f,bs)
      NLL= lr_model_nll(β_curr,x,y;σ=σ_model)
      f(β)=lr_model_nll(β,x,y;σ=σ_model)[1]
      grad_β = gradient(f,β_curr)[1]
      #@info "loss: $(NLL)  β: $β_curr grad:$grad_β   " #TODO: log loss, if you want
to monitor training progress
      β_curr = β_curr + grad_β * lr
    end
    return β_curr
end
```

```
train_lin_reg (generic function with 1 method)
```

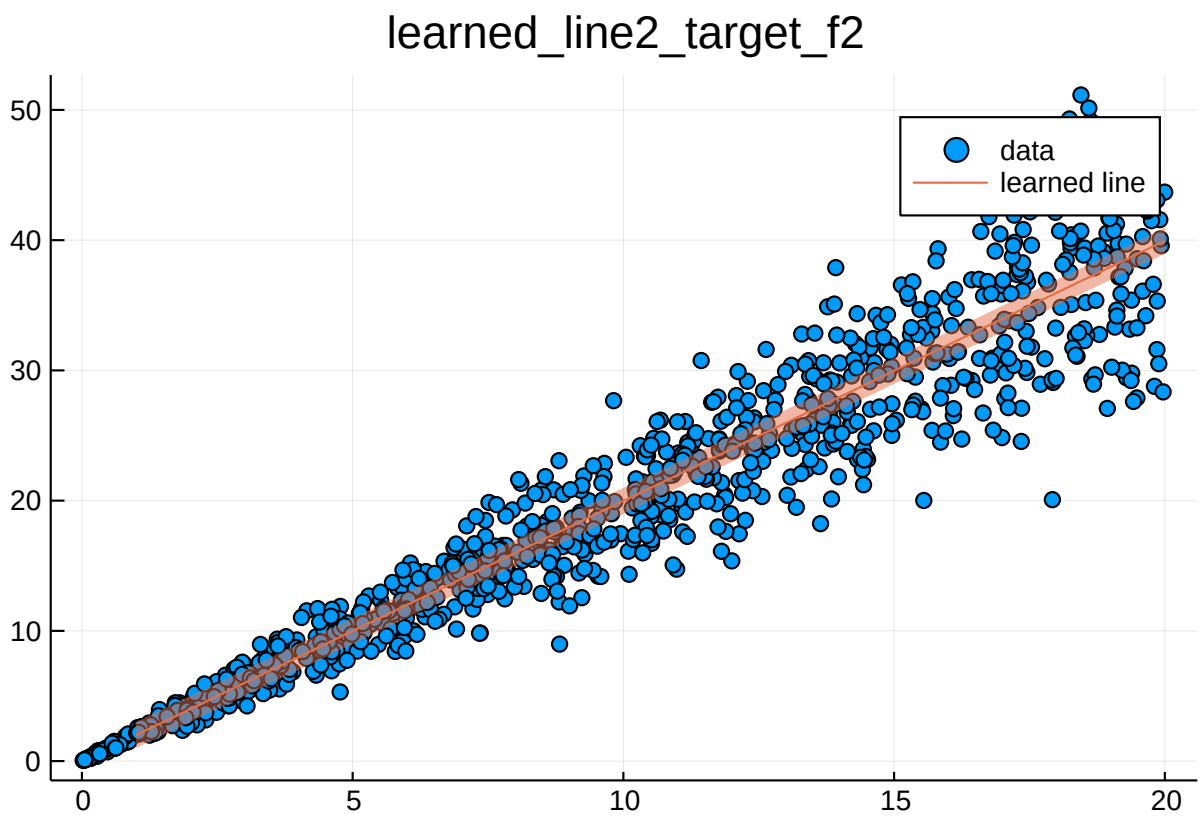
2. [2pts] For each target function, start with an initial parameter β, learn an estimate for β_{learned} by gradient descent. Then plot a $n = 1000$ sample of the data and the learned linear regression model with shaded region for uncertainty corresponding to plus/minus one standard deviation.

Again, use the derived formular for the negative log-likelihood function, β initial value can be set at very wide random range. If use gaussian log-likelihood function, then β initial can only be set at 2+randn(), which feels like cheating, as we assume the beta value almost known.

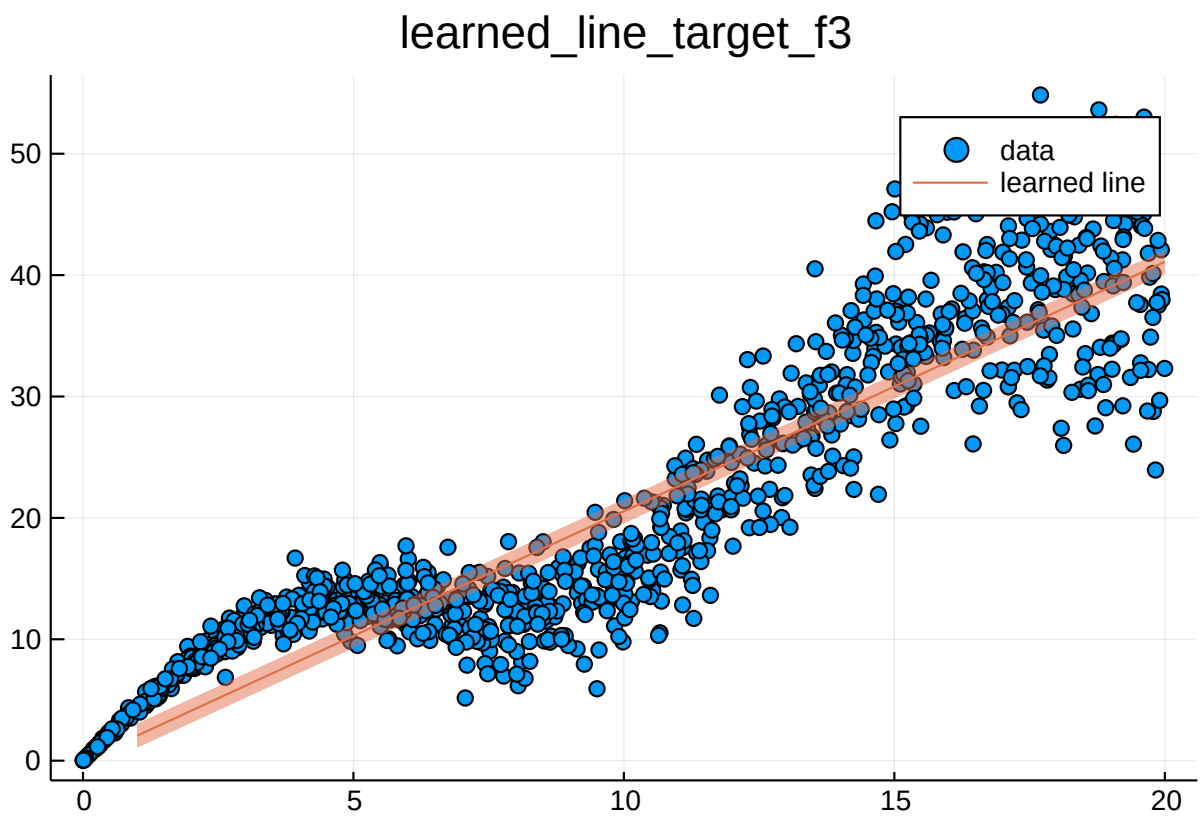
```
# Plot for learned regression for target_f1
β_init = 1000*randn() # Initial parameter
β1_learned= train_lin_reg(target_f1, β_init; bs= 100, lr = 1e-6, iters=1000, σ_model = 1.
)
x_range=1:20
learned_line1 = β1_learned .* x_range
x,y = sample_batch(target_f1,1000)
plotl1 = plot(x',y,title="learned_line_target_f1",seriestype=:scatter,label="data")
display(plot!(plotl1,learned_line1,label="learned line",ribbon=1))
```




```
#Plot for learned regression for target_f2
β_init2 = 1000*randn()
β2_learned = train_lin_reg(target_f2, β_init2; bs= 100, lr = 1e-6, iters=1000, σ_model =
1. )
learned_line2 = β2_learned .* x_range
x,y = sample_batch(target_f2,1000)
plotl2 = plot(x',y,title="learned_line2_target_f2",seriestype=:scatter,label="data")
display(plot!(plotl2,learned_line2,ribbon=1,label="learned line"))
```




```
#Plot for learned regression for target_f3
β_init3 = 1000*randn()
β3_learned = train_lin_reg(target_f3, β_init3; bs= 100, lr = 1e-6, iters=1000, σ_model =
1. )
learned_line3 = β3_learned .* x_range
x,y = sample_batch(target_f3,1000)
plotl3 = plot(x',y,title="learned_line_target_f3",seriestype=:scatter,label="data")
display(plot!(plotl3,learned_line3,ribbon=1,label="learned line"))
```



2.5.2 Non-linear Regression with a Neural Network [9pts]

In the previous questions we have considered a linear regression model

$$Y \sim \mathcal{N}(X^T\beta, \sigma^2)$$

This model specified the mean of the predictive distribution for each datapoint by the product of that datapoint with our parameter.

Now, let us generalize this to consider a model where the mean of the predictive distribution is a non-linear function of each datapoint. We will have our non-linear model be a simple function called `neural_net` with parameters θ (collection of weights and biases).

$$Y \sim \mathcal{N}(\texttt{neural\_net}(X, \theta), \sigma^2)$$

1. [3pts] Write the code for a fully-connected neural network (multi-layer perceptron) with one 10-dimensional hidden layer and a `tanh` nonlinearirty. You must write this yourself using only basic operations like matrix multiply and `tanh`, you may not use layers provided by a library.

   This network will output the mean vector, test that it outputs the correct shape for some random parameters.

```
function neural_net(x,θ)
    h=tanh.(θ.w1'*x .+ θ.b1)
    z=θ.w2'*h .+ θ.b2
return vec(z ./size(x)[2])
end
```

```
# Random initial Parameters
w1=30*randn(1,10)
b1=10*randn(10,1)
w2=30*randn(10,1)
b2=10*randn()
θ = (w1=w1, b1=b1, w2=w2, b2=b2)

@testset "neural net mean vector output" begin
n = 100
x,y = sample_batch(target_f1,n)
μ = neural_net(x,θ)
@test size(μ) == (n,)
end
```

```
Test Summary:                  | Pass  Total
neural net mean vector output |    1      1
Test.DefaultTestSet("neural net mean vector output", Any[], 1, false)
```

2. [2pts] Write the code that computes the negative log-likelihood for this model where the mean is given by the output of the neural network and $\sigma = 1.0$

In this case there is no derived formular for negative log-likelihood, so I have no confidence it will not produce NaN all the time.

```
function nn_model_nll(θ,x,y;σ=1)
  return sum(-gaussian_log_likelihood(θ,σ,y))
end
```

```
nn_model_nll (generic function with 1 method)
```

3. [2pts] Write a function `train_nn_reg` that accepts a target function and an initial estimate for θ and some hyperparameters for batch-size, model variance, learning rate, and number of iterations. Then, for each iteration:

   - sample data from the target function
   - compute gradients of negative log-likelihood with respect to θ
   - update the estimate of θ with gradient descent with specified learning rate

   and, after all iterations, returns the final estimate of θ.

```
using Logging # Print training progress to REPL, not pdf

function train_nn_reg(target_f, θ_init; bs= 100, lr = 1e-5, iters=1000, σ_model = 1. )
    θ_curr = θ_init
    w1=4000*randn(1,10)
    b1=1000*randn(10,1)
    w2=4000*randn(10,1)
    b2=1000*randn()
    theta= (w1=w1, b1=b1, w2=w2, b2=b2)
    for i in 1:iters
      x,y = sample_batch(target_f, bs)
      μ=neural_net(x,theta)
      NLL= nn_model_nll(μ,x,y;σ=σ_model)
```

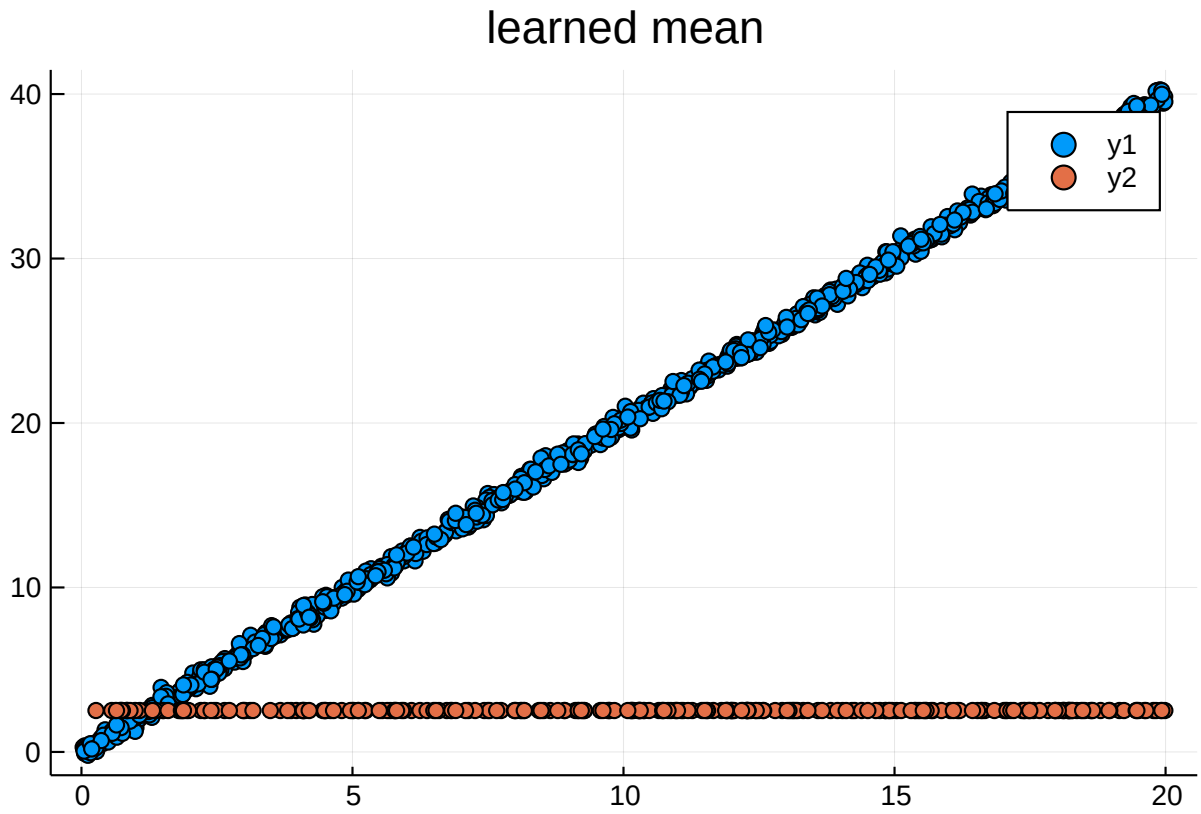
```
        f(θ)=nn_model_nll(θ,x,y;σ=σ_model)
        grad_θ = gradient(f,θ_curr)[1]
        #@info "loss: $(NLL), θ: $θ_curr, grad: $grad_θ " #TODO: log loss, if you want
to montior training
        θ_curr = θ_curr - grad_θ*lr
    end
    return θ_curr
end
```

```
train_nn_reg (generic function with 1 method)
```

4. [2pts] For each target function, start with an initialization of the network parameters, θ, use your train function to minimize the negative log-likelihood and find an estimate for θ_{learned} by gradient descent. Then plot a $n = 1000$ sample of the data and the learned regression model with shaded uncertainty bounds given by $\sigma = 1.0$

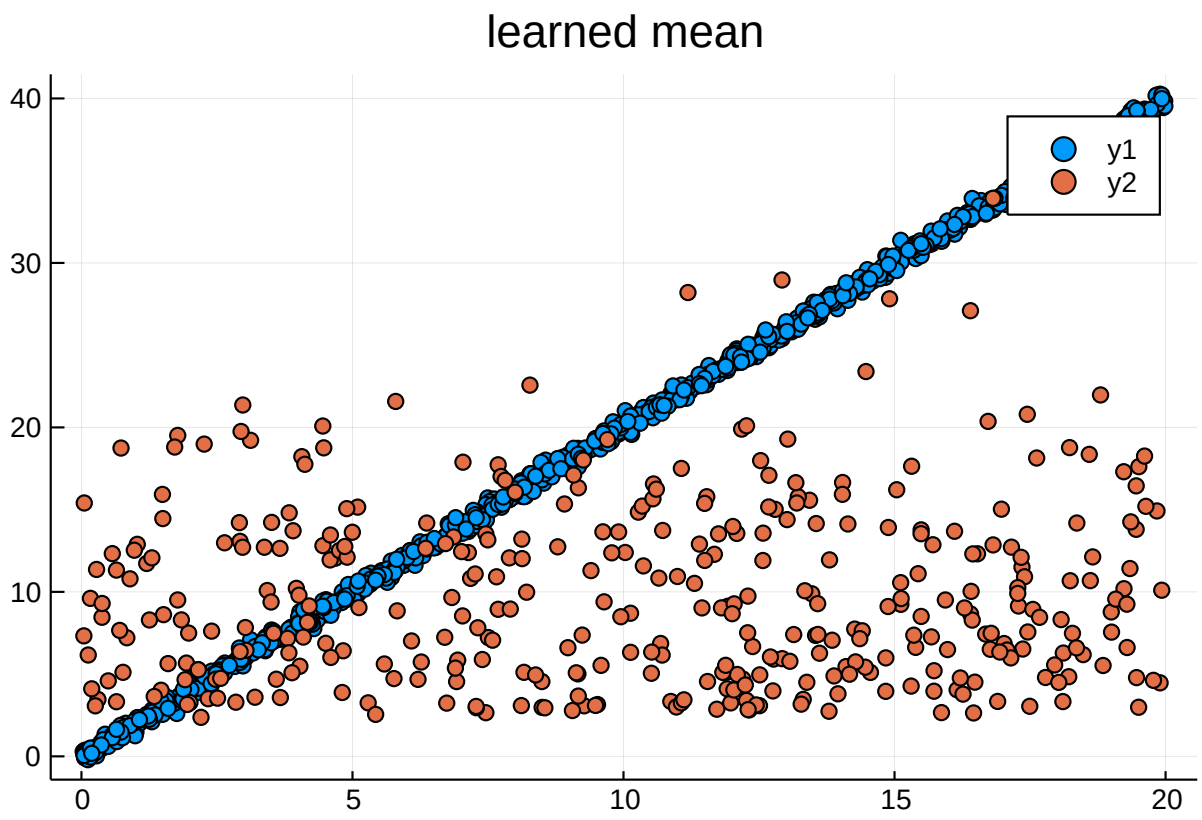
The correct way to set θ_init is use neural network to generate initial mean, but as the initial value unpredicatable, some times, the process will not generate any result at all.

```
w1=4000*randn(1,10)
b1=1000*randn(10,1)
w2=4000*randn(10,1)
b2=1000*randn()
theta= (w1=w1, b1=b1, w2=w2, b2=b2)
x,y = sample_batch(target_f1,1000)
θ_init=neural_net(x,theta)
θ1_learned = train_nn_reg(target_f1, θ_init; bs= 1000, lr = 1e-6, iters=1000, σ_model = 1.
)
display(plot(x',hcat(y,θ1_learned),title="learned mean",seriestype=:scatter))
```

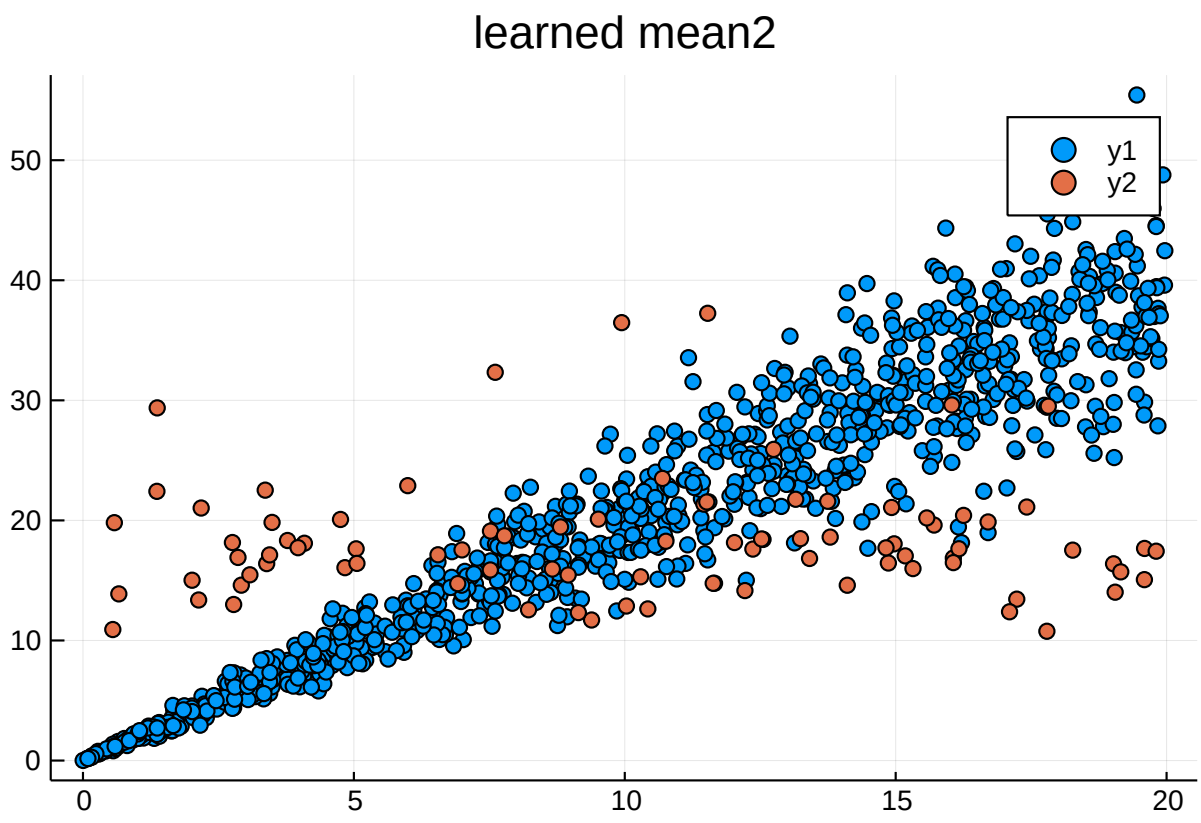


Use narrowed range of initial mean value:

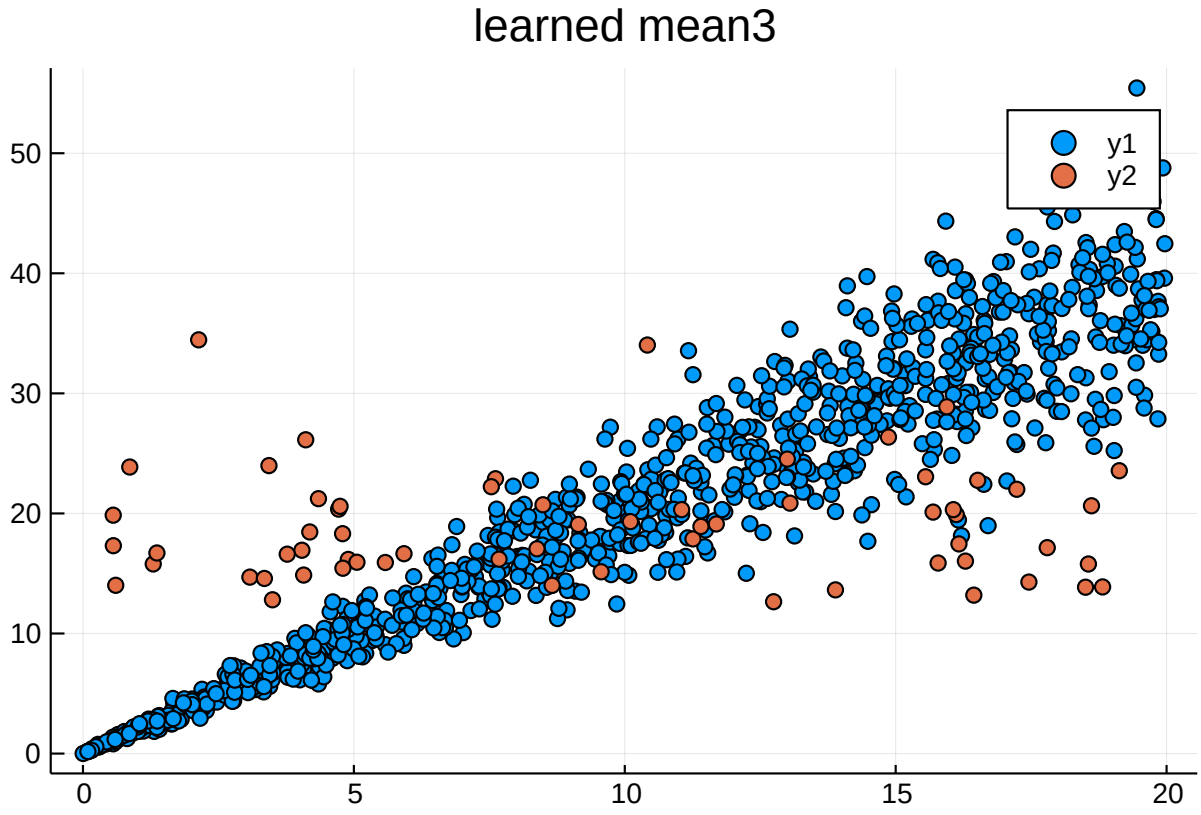
```
θ_init=10*randn(1000)
θ1_learned = train_nn_reg(target_f1, θ_init; bs= 1000, lr = 1e-6, iters=1000, σ_model = 1.
)
display(plot(x',hcat(y,θ1_learned),title="learned mean",seriestype=:scatter))
```




```
θ_init=10*randn(1000)
θ2_learned = train_nn_reg(target_f2, θ_init; bs= 1000, lr = 1e-5, iters=1000, σ_model = 1.
)
x,y = sample_batch(target_f2,1000)
display(plot(x',hcat(y,θ2_learned),title="learned mean2",seriestype=:scatter))
```




```
θ_init=10*randn(1000)
θ3_learned = train_nn_reg(target_f3, θ_init; bs= 1000, lr = 1e-5, iters=1000, σ_model = 1.
)
plott3 = plot(x',hcat(y,θ3_learned),title="learned mean3",seriestype=:scatter)
```



2.5.3 Non-linear Regression and Input-dependent Variance with a Neural Network [8pts]

In the previous questions we've gone from a gaussian model with mean given by linear combination

$$Y \sim \mathcal{N}(X^T\beta, \sigma^2)$$

to gaussian model with mean given by non-linear function of the data (neural network)

$$Y \sim \mathcal{N}(\texttt{neural\_net}(X, \theta), \sigma^2)$$

However, in all cases we have considered so far, we specify a fixed variance for our model distribution. We know that two of our target datasets have heteroscedastic noise, meaning any fixed choice of variance will poorly model the data.

In this question we will use a neural network to learn both the mean and log-variance of our gaussian model.

$$\mu, \log\sigma = \texttt{neural\_net}(X, \theta)$$
$$Y \sim \mathcal{N}(\mu, \exp(\log\sigma)^2)$$

1. [1pts] Write the code for a fully-connected neural network (multi-layer perceptron) with one 10-dimensional hidden layer and a `tanh` nonlinearirty, and outputs both a vector for mean and $\log\sigma$. Test the output shape is as expected.

```
# Neural Network Function
function neural_net_w_var(x,θ)
  h = tanh.(θ.w1'*x .+ θ.b1)
  z = θ.w2'*h .+ θ.b2
  μ = vec(z ./size(x)[2])
  logσ = log.(1 /(size(x)[2]-1) .* (vec(x)-μ).^2)
  return μ, logσ
end

# Random initial Parameters
θ = #TODO
@testset "neural net mean and logsigma vector output" begin
n = 100
x,y = sample_batch(target_f1,n)
μ, logσ = neural_net_w_var(x,θ)
@test size(μ) == (n,)
@test size(logσ) == (n,)
end
```

```
Test Summary:                                | Pass  Total
neural net mean and logsigma vector output |    2      2
Test.DefaultTestSet("neural net mean and logsigma vector output", Any[], 2,
 false)
```

2. [2pts] Write the code that computes the negative log-likelihood for this model where the mean and $\log\sigma$ is given by the output of the neural network. (Hint: Don't forget to take $\exp\log\sigma$)

```
function nn_with_var_model_nll(θ,x,y)
  μ=θ[1]
  σ=exp.(θ[2])
  return sum(-gaussian_log_likelihood(μ,σ,y))
end
```

```
nn_with_var_model_nll (generic function with 1 method)
```

3. [1pts] Write a function `train_nn_w_var_reg` that accepts a target function and an initial estimate for θ and some hyperparameters for batch-size, learning rate, and number of iterations. Then, for each iteration:

   - sample data from the target function
   - compute gradients of negative log-likelihood with respect to θ
   - update the estimate of θ with gradient descent with specified learning rate

   and, after all iterations, returns the final estimate of θ.

```
function train_nn_w_var_reg(target_f, θ_init; bs= 100, lr = 1e-4, iters=10000)
  θ_curr = θ_init
  w1=4000*randn(1,10)
  b1=1000*randn(10,1)
  w2=4000*randn(10,1)
  b2=1000*randn()
  theta= (w1=w1, b1=b1, w2=w2, b2=b2)
  for i in 1:iters
    x,y = sample_batch(target_f,bs)
    θ = neural_net_w_var(x,theta)
    NLL1= nn_with_var_model_nll(θ,x,y)
    f(θ)=nn_with_var_model_nll(θ,x,y)
    #@info "loss: $(NLL), θ: $θ_curr, grad: $grad_θ "
    grad_θ = gradient(f, θ_curr)[1]
    θ_curr = θ_curr .- grad_θ*lr
  end
  return θ_curr
end
```

```
train_nn_w_var_reg (generic function with 1 method)
```

4. [4pts] For each target function, start with an initialization of the network parameters, θ, learn an estimate for θ_{learned} by gradient descent. Then plot a $n = 1000$ sample of the dataset and the learned regression model with shaded uncertainty bounds corresponding to plus/minus one standard deviation given by the variance of the predictive distribution at each input location (output by the neural network). (Hint: `ribbon` argument for shaded uncertainty bounds can accept a vector of σ)

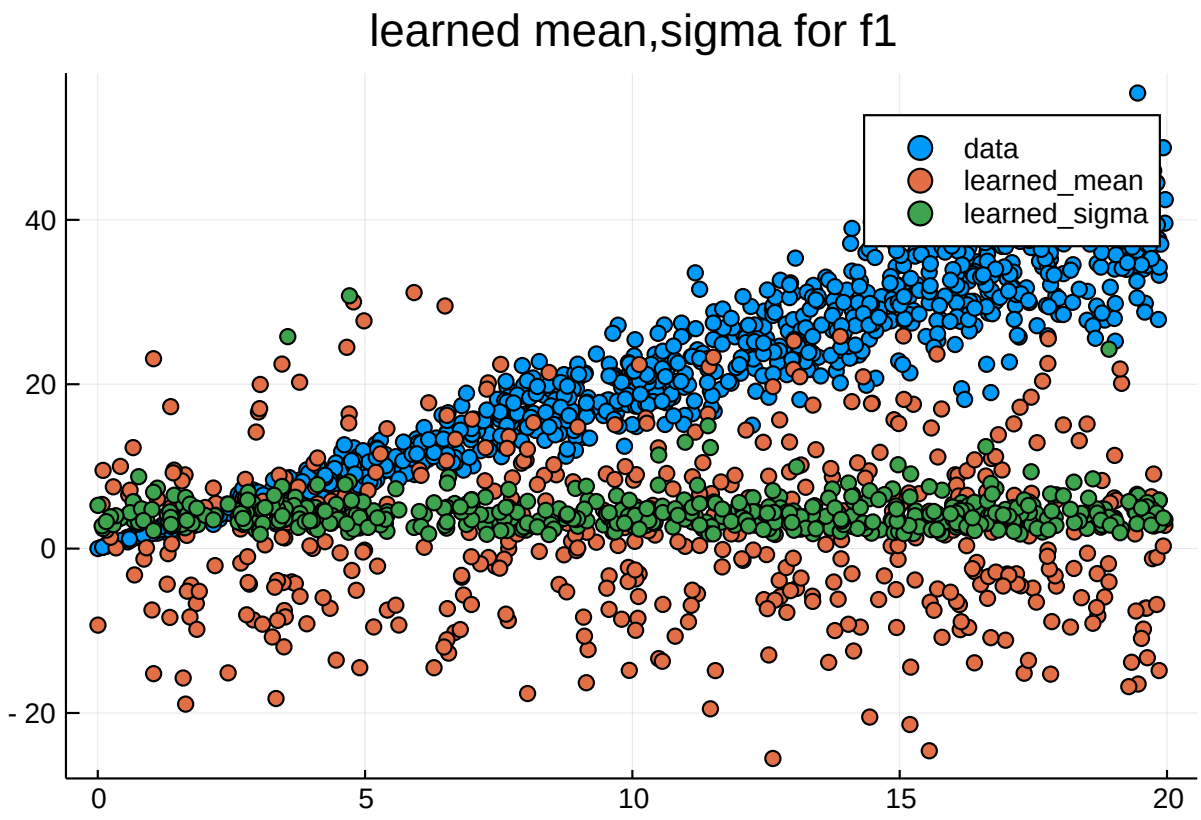
   Note: Learning the variance is tricky, and this may be unstable during training. There are some things you can try:

   - Adjusting the hyperparameters like learning rate and batch size
   - Train for more iterations
   - Try a different random initialization, like sample random weights and bias matrices with lower variance.

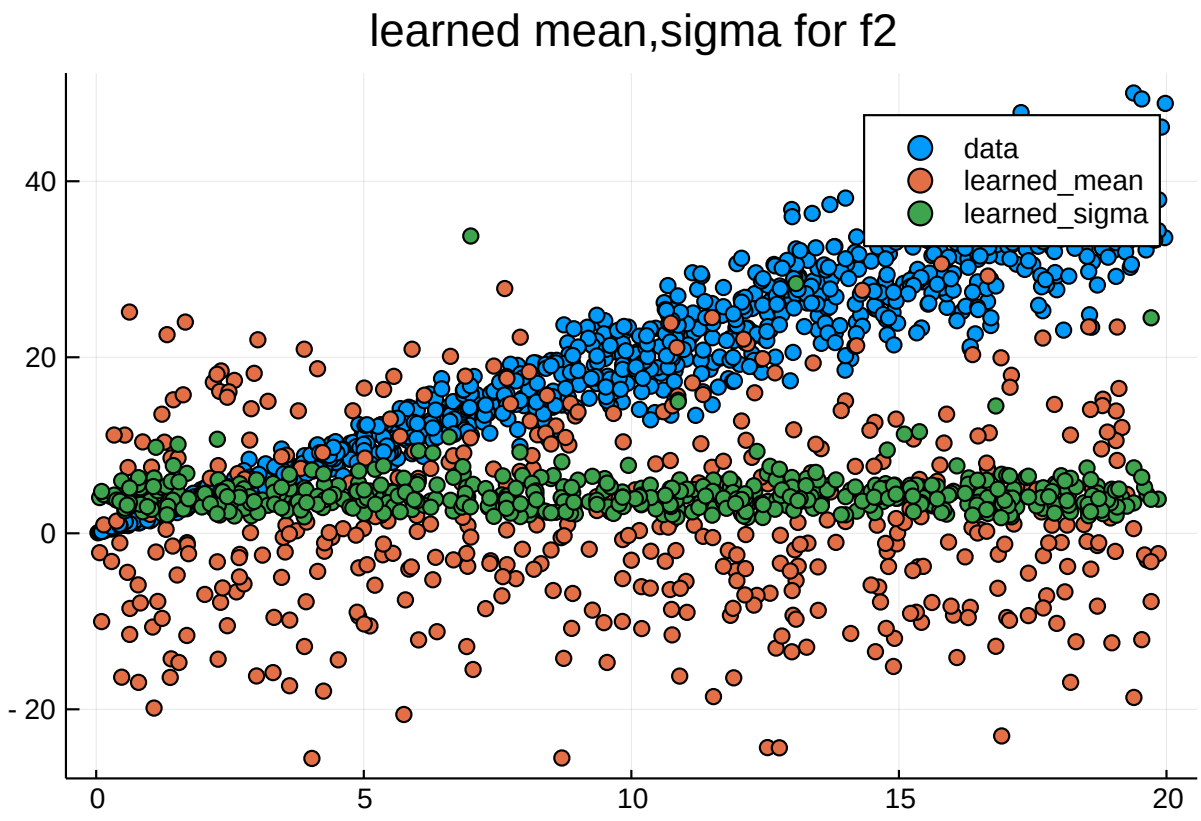
For this question **you will not be assessed on the final quality of your model**. Specifically, if you fails to train an optimal model for the data that is okay. You are expected to learn something that is somewhat reasonable, and **demonstrates that this model is training and learning variance**.

If your implementation is correct, it is possible to learn a reasonable model with fewer than 10 minutes of training on a laptop CPU. The default hyperparameters should help, but may need some tuning.

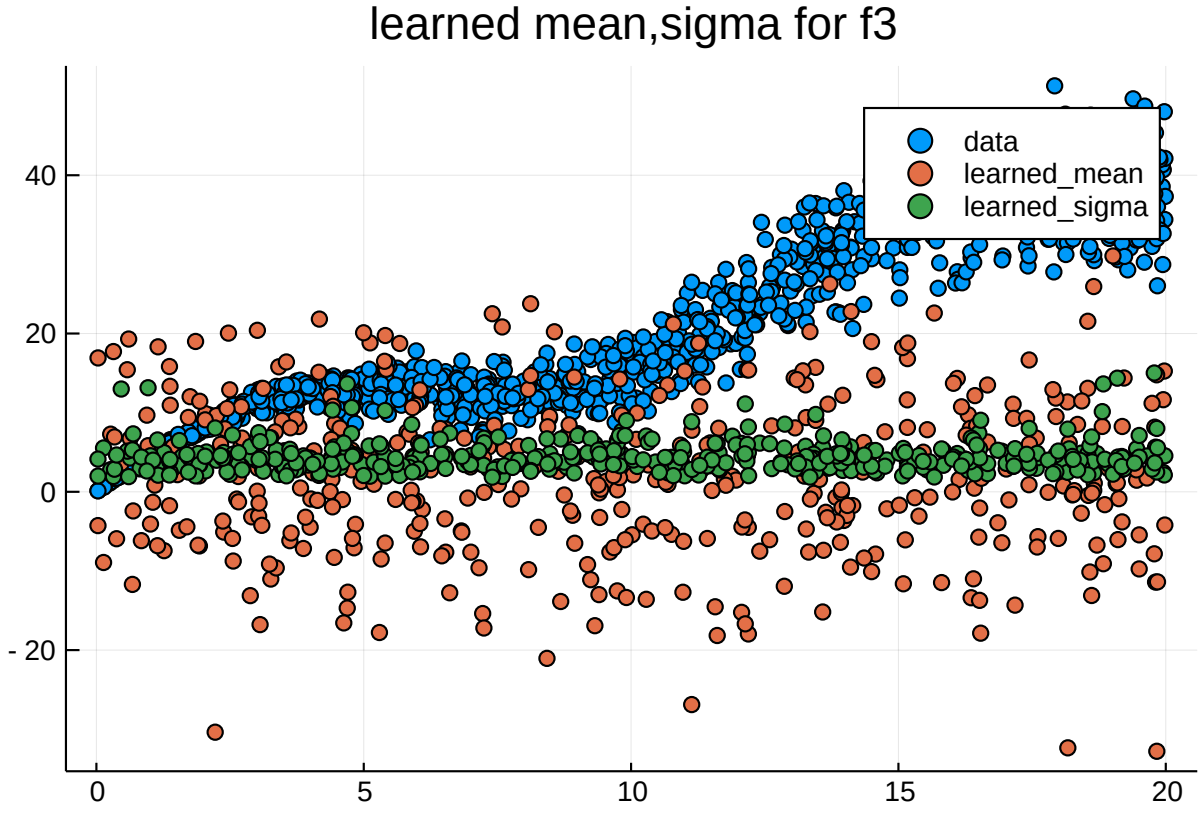
```
θ_init=[10*randn(1000),randn(1000)]
θ1_learned = train_nn_w_var_reg(target_f1, θ_init; bs= 1000, lr = 1e-5, iters=1000)
sigma1= exp.(θ1_learned[2])
display(plot(x',hcat(y,θ1_learned[1],sigma1),title="learned mean,sigma for f1",
seriestype=:scatter,label=["data" "learned_mean" "learned_sigma"]))
```




```
θ_init=[10*randn(1000),randn(1000)]
θ2_learned = train_nn_w_var_reg(target_f2, θ_init; bs= 1000, lr = 1e-5, iters=1000)
sigma2=exp.(θ2_learned[2])
x,y = sample_batch(target_f2,1000)
display(plot(x',hcat(y,θ2_learned[1],sigma2),title="learned mean,sigma for
f2",seriestype=:scatter,label=["data" "learned_mean" "learned_sigma"]))
```




```
θ_init=[10*randn(1000),randn(1000)]
θ3_learned = train_nn_w_var_reg(target_f3, θ_init; bs= 1000, lr = 1e-5, iters=1000)
sigma3=exp.(θ3_learned[2])
x,y = sample_batch(target_f3,1000)
plott3 = plot(x',hcat(y,θ3_learned[1],sigma3),title="learned mean,sigma for
f3",seriestype=:scatter, label=["data" "learned_mean" "learned_sigma"])
```



If you would like to take the time to train a very good model of the data (specifically for target functions 2 and 3) with a neural network that outputs both mean and $\log \sigma$ you can do this, but it is not necessary to achieve full marks. You can try

- Using a more stable optimizer, like Adam. You may import this from a library.
- Increasing the expressivity of the neural network, increase the number of layers or the dimensionality of the hidden layer.
- Careful tuning of hyperparameters, like learning rate and batchsize.