Assignment 1

Hanyu Qi and 1003046250

February 24, 2020

```
using Weave
weave("/Users/mac/Downloads/STA414-2020-A1-skyqqq-master/A1.jmd", doctype = "md2pdf")
```

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.

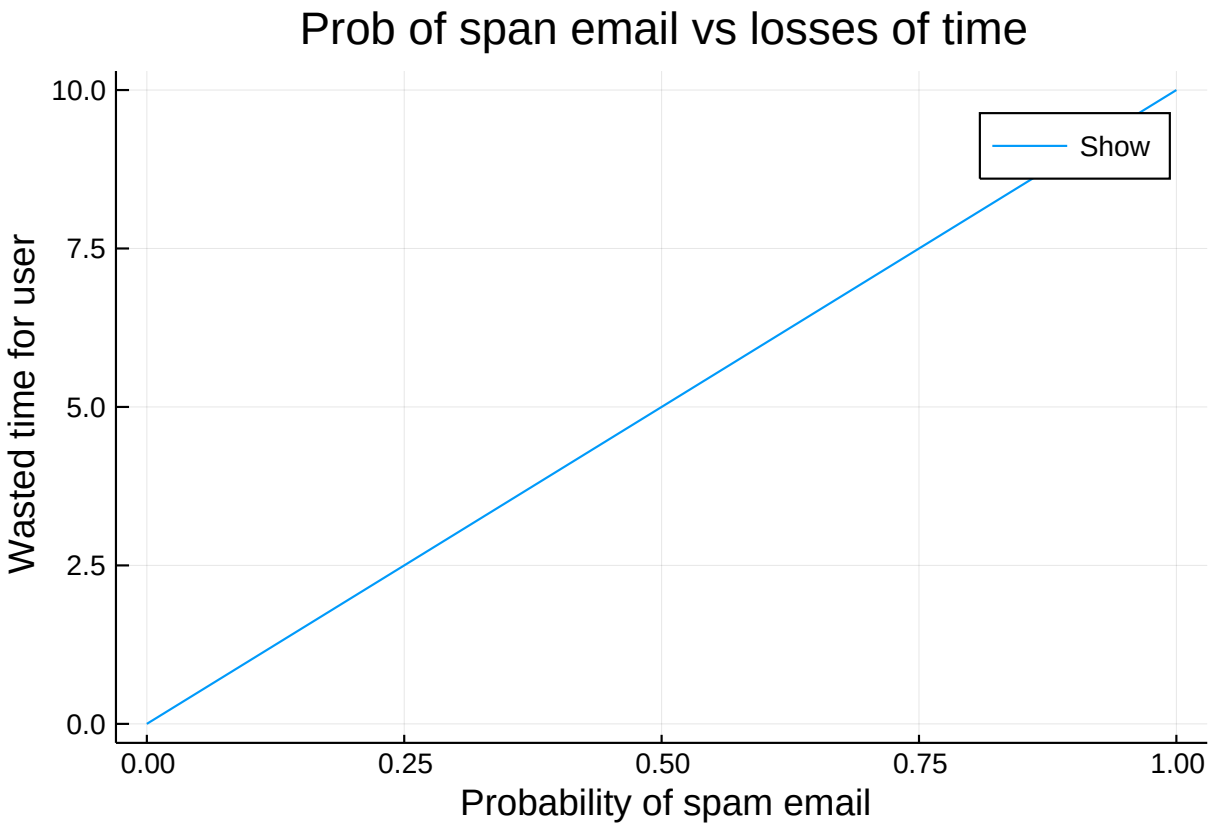
Depending on whether or not the email really is spam, the user will suffer a different amount of wasted time for the different actions we can take, $L(\text{action}, \text{spam})$:

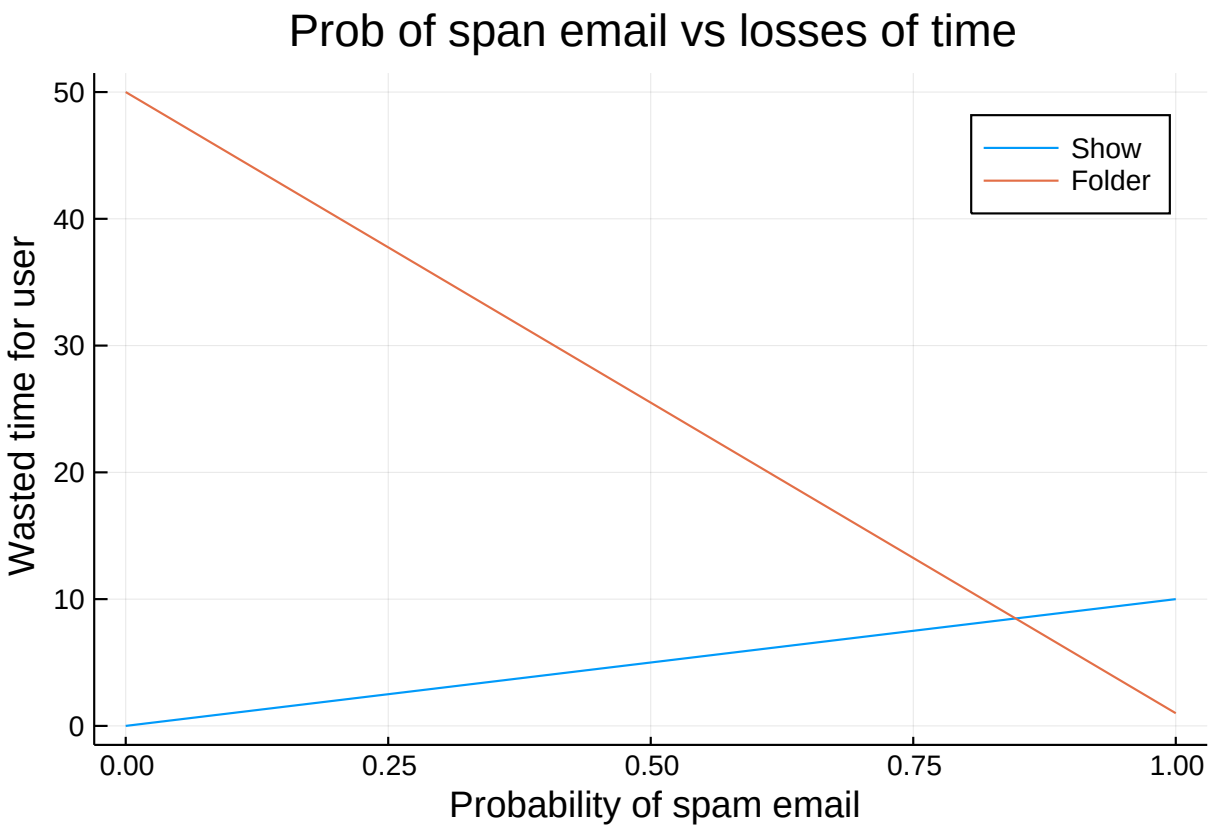
Action	Spam	Not spam
Show	10	0
Folder	1	50
Delete	0	200

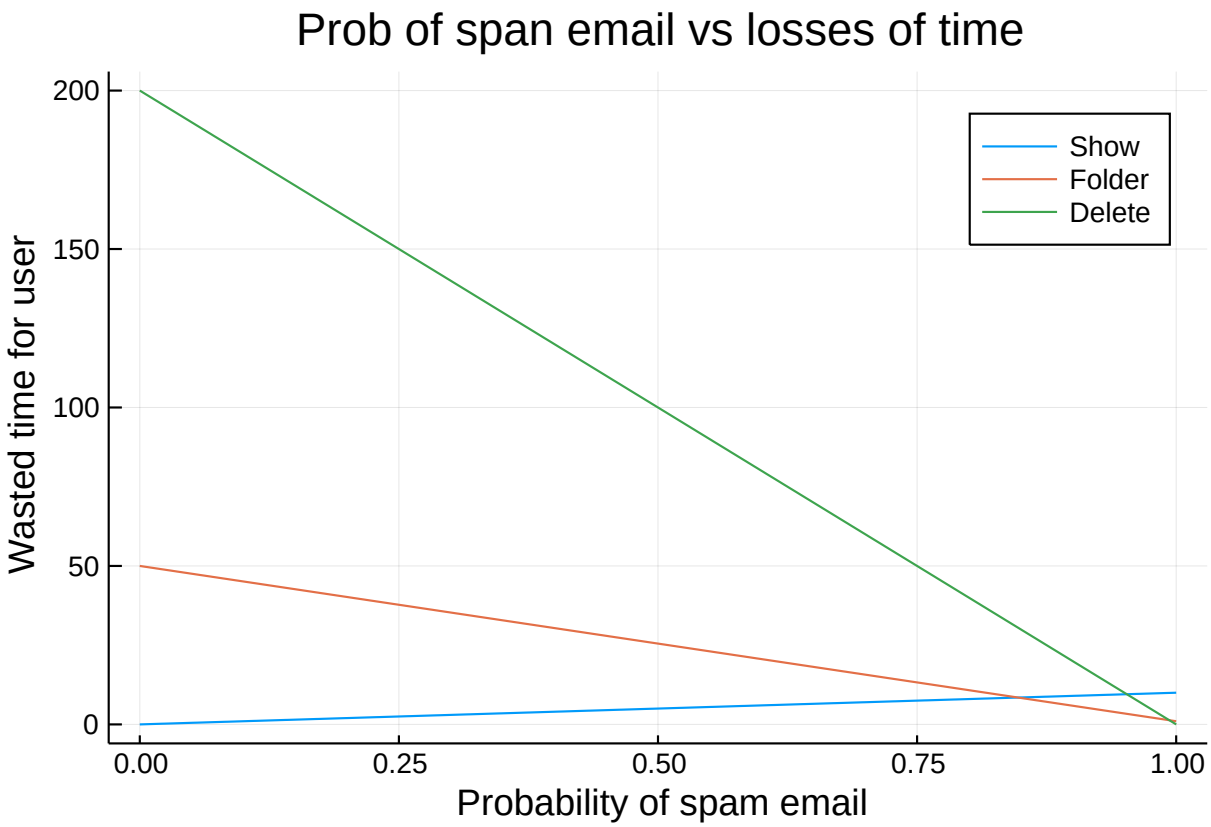
1. [3pts] Plot the expected wasted user time for each of the three possible actions, as a function of the probability of spam: $p(\text{spam}|\text{email})$

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

num_actions = length(losses)

function expected_loss_of_action(prob_spam, action)
    expect = []
    for j in 1:length(prob_spam)
      result = losses[action][1] * prob_spam[j] + losses[action][2] * (1-prob_spam[j])
      append!(expect,result)
    end
```

```
    return expect
end

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

# Make plot
using Plots
plot()
xlabel!("Probability of spam email")
ylabel!("Wasted time for user")
title!("Prob of span email vs losses of time")
labels = ["Show","Folder","Delete"]
for action in 1:num_actions
  display(plot!(prob_range, expected_loss_of_action(prob_range, action),label =
labels[action]))
end
```







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

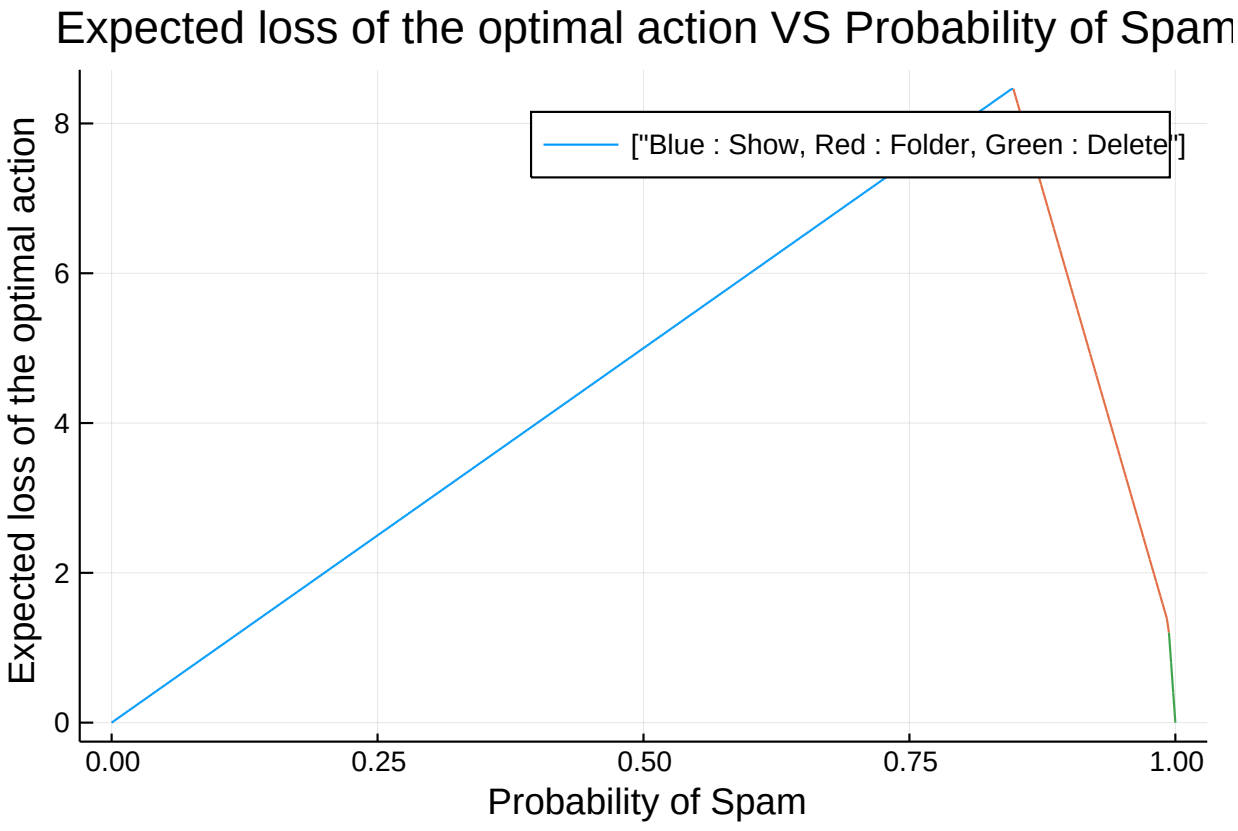
```
labels = ["Show", "Folder","Delete"]
function optimal_action(prob_spam)
  results = []
```

```
  for action in 1:length(labels)
    result = losses[action][1] * prob_spam + losses[action][2] * (1-prob_spam)
    append!(results,result)
  end
  return findmin(results)[2]
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)
best_losses = []
best_actions = []
for p in prob_range
  action = optimal_action(p)
  loss = losses[action][1] * p + losses[action][2]*(1-p)
  append!(best_losses,loss)
  append!(best_actions,action)
end

plot()
xlabel!("Probability of Spam")
ylabel!("Expected loss of the optimal action")
title!("Expected loss of the optimal action VS Probability of Spam")
plot!(prob_range, best_losses, linecolor=best_actions, label = ["Blue : Show, Red :
Folder, Green : Delete"])
```



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.

To get the range of the probabilities of an email being spam should we delete, we need to solve the probability at the interception of Red and Green lines. p+50(1-p) = 200(1-p) so we can get p = 150/151

So the exactly probability of an email being spam we should delete is $\frac{150}{151}$

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 $n < m$?

When $n < m$, X is not full rank because there are n-m variables can be expressed by the rest of n variables so some columns are linearly dependent.

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

$$\begin{aligned} E(\hat{\beta}) &= E((XX^T)^{-1}XY) && (1)\\ &= (XX^T)^{-1}XE(Y) && (2)\\ &= (XX^T)^{-1}XX^T\beta && (3)\\ &= \beta && (4) \end{aligned}$$

$$\begin{aligned} Cov(\hat{\beta}) &= Cov((XX^T)^{-1}XY) && (5)\\ &= (XX^T)^{-1}XCov(Y)((XX^T)^{-1}X)^T && (6)\\ &= Cov(Y)(XX^T)^{-1}XX^T(XX^T)^{-1} && (7)\\ &= \sigma^2(XX^T)^{-1} && (8) \end{aligned}$$

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 $\sum_{i=1}^n a_i^2 = a^T a$]

Since $Y \sim \mathcal{N}(X^T\beta, \sigma^2 I)$ so we know $y_i \sim \mathcal{N}(x_i\beta, \sigma^2)$

And $f_Y(y_i) = \frac{1}{\sqrt{2\pi}\sigma} exp(-\frac{1}{2\sigma^2}(y_i - x_i\beta)^2)$

We know $l = \sum_{i=1}^{n} \ln(\frac{1}{\sqrt{2\pi}\sigma}) - \frac{1}{2\sigma^2}(y_i - x_i\beta)^2$

So $l = n\ln(\frac{1}{\sqrt{2\pi}\sigma}) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - x_i\beta)^2$

To maximize the likelihood function we need we maximize the log-likelihood function , since there is a negative sign in front of the squared error. So it is equivalent to minimize the squared error.

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]

$$\begin{aligned}\sum_{i=1}^{n}(y_i - x_i\beta)^2 &= \sum_{i=1}^{n}(y_i)^2 - 2\sum_{i=1}^{n}(x_i y_i \beta) + \sum_{i=1}^{n}(x_i)^2\beta^2 \\ &= y^T y - 2y^T x^T \beta + \beta^T x x^T \beta \\ &= y^T y - y^T x^T \beta - \beta^T x y + \beta^T x x^T \beta \\ &= (y - x^T\beta)^T(y - x^T\beta)\end{aligned}$$

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.

Since $y_i \sim \mathcal{N}(x_i\beta, \sigma^2)$ we can write its pdf as $f_Y(y_i) = \frac{1}{\sqrt{2\pi}\sigma} exp(-\frac{1}{2\sigma^2}(y_i - x_i\beta)^2)$ We know $l = n\ln(\frac{1}{\sqrt{2\pi}\sigma}) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - x_i\beta)^2$ So $-l = \frac{n}{2}\ln 2\pi + n\ln\sigma + \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - x_i\beta)^2$ We can write $\frac{\partial -l}{\partial \beta} = \frac{-1}{\sigma^2}\sum_{i=1}^{n}(x_i y_i) + \sum_{i=1}^{n}(x_i)^2\beta = 0$ after seting to 0 Then $\hat{\beta} = \frac{\sum_{i=1}^{n} x_i y_i}{\sum_{i=1}^{n} x_i x_i} = (XX^T)^{-1}XY$

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

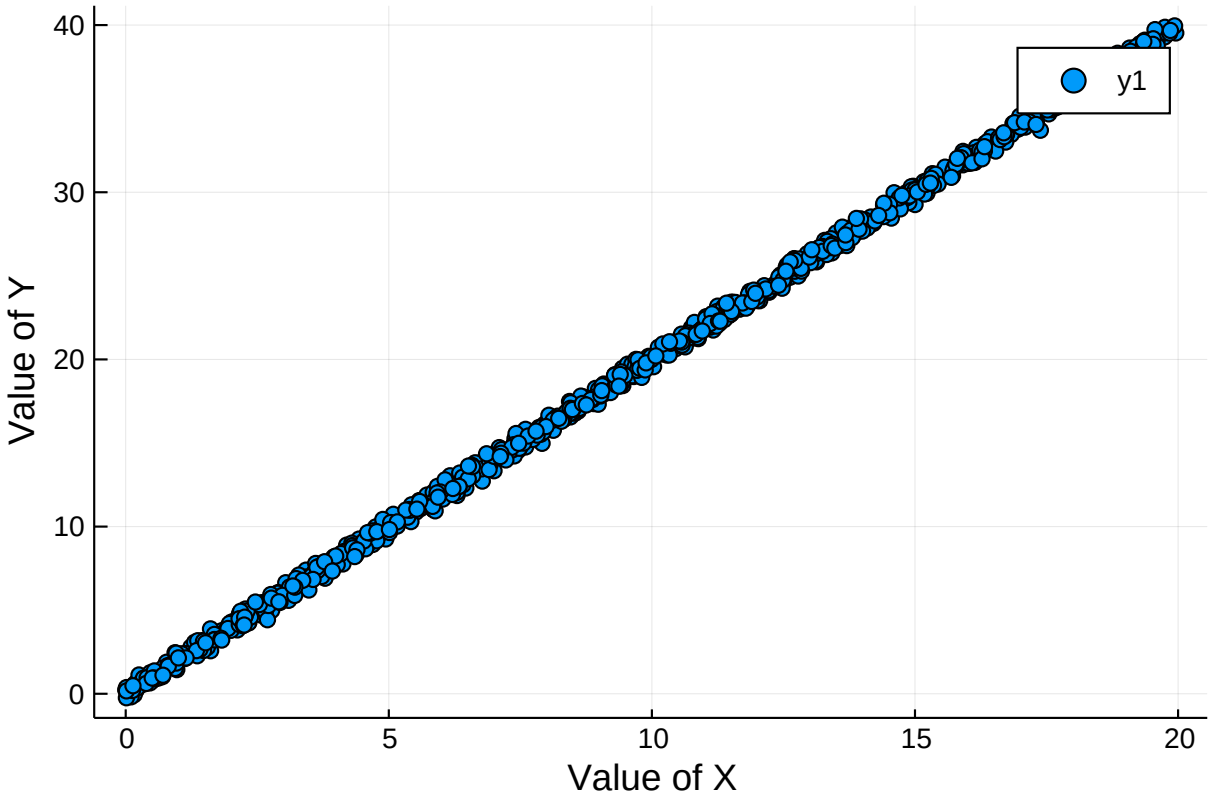
```
sample_batch (generic function with 1 method)
```

```
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
```

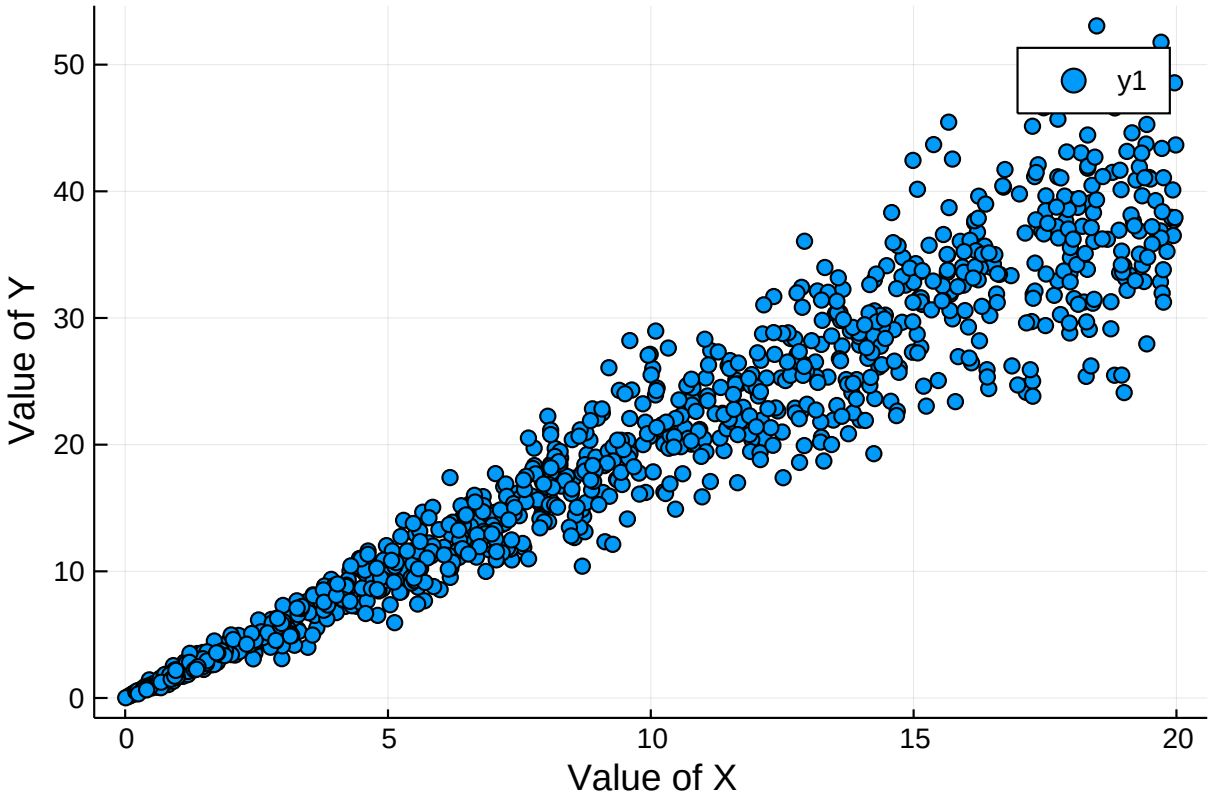
```
Test Summary:                  | Pass  Total
sample dimensions are correct |    6      6
Test.DefaultTestSet("sample dimensions are correct", Any[], 6, false)
```

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.**

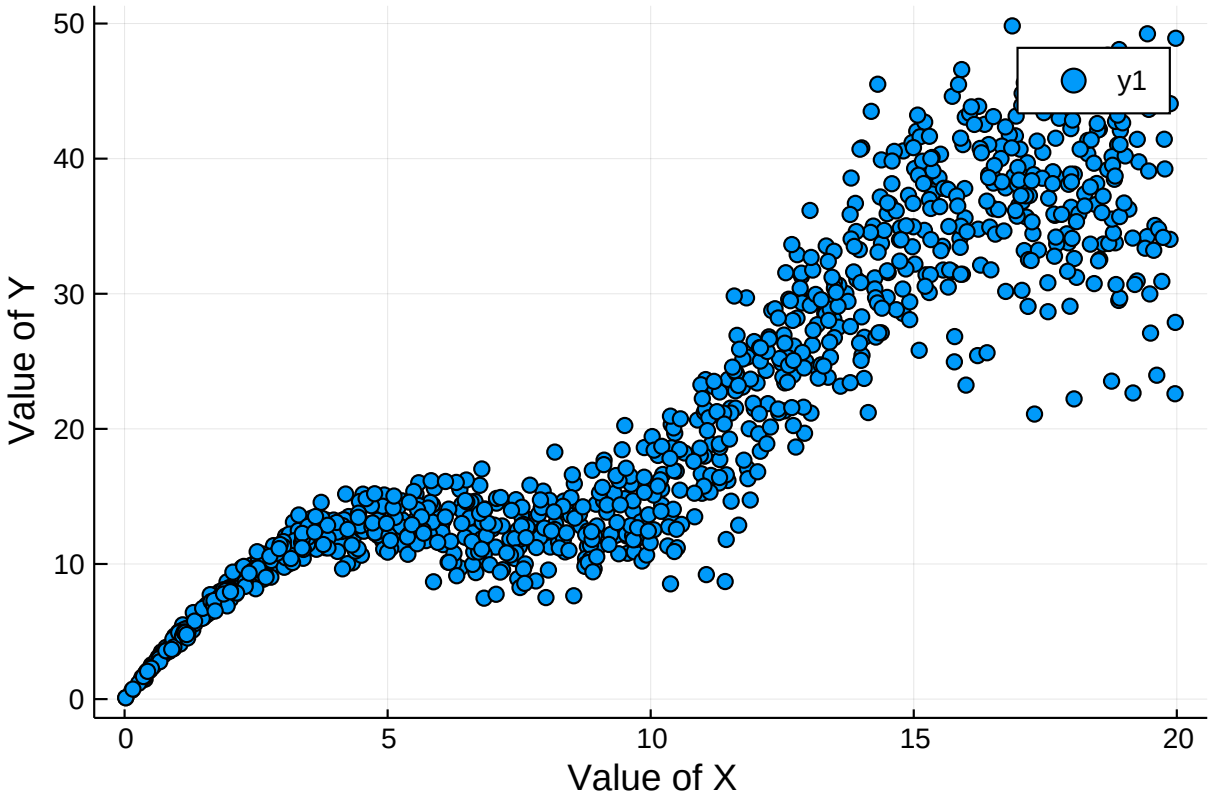
```
using Plots
using Distributions
using Random

x1,y1 = sample_batch(target_f1,1000)
plot()
xlabel!("Value of X")
ylabel!("Value of Y")
plot_f1 = scatter!((x1)',y1)
```




```
using Plots
using Distributions
using Random
x2,y2 = sample_batch(target_f2,1000)
plot()
xlabel!("Value of X")
ylabel!("Value of Y")
plot_f2 = scatter!((x2)',y2)
```




```
using Plots
using Distributions
using Random
x3,y3 = sample_batch(target_f3,1000)
plot()
xlabel!("Value of X")
ylabel!("Value of Y")
plot_f3 = scatter!((x3)',y3)
```



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

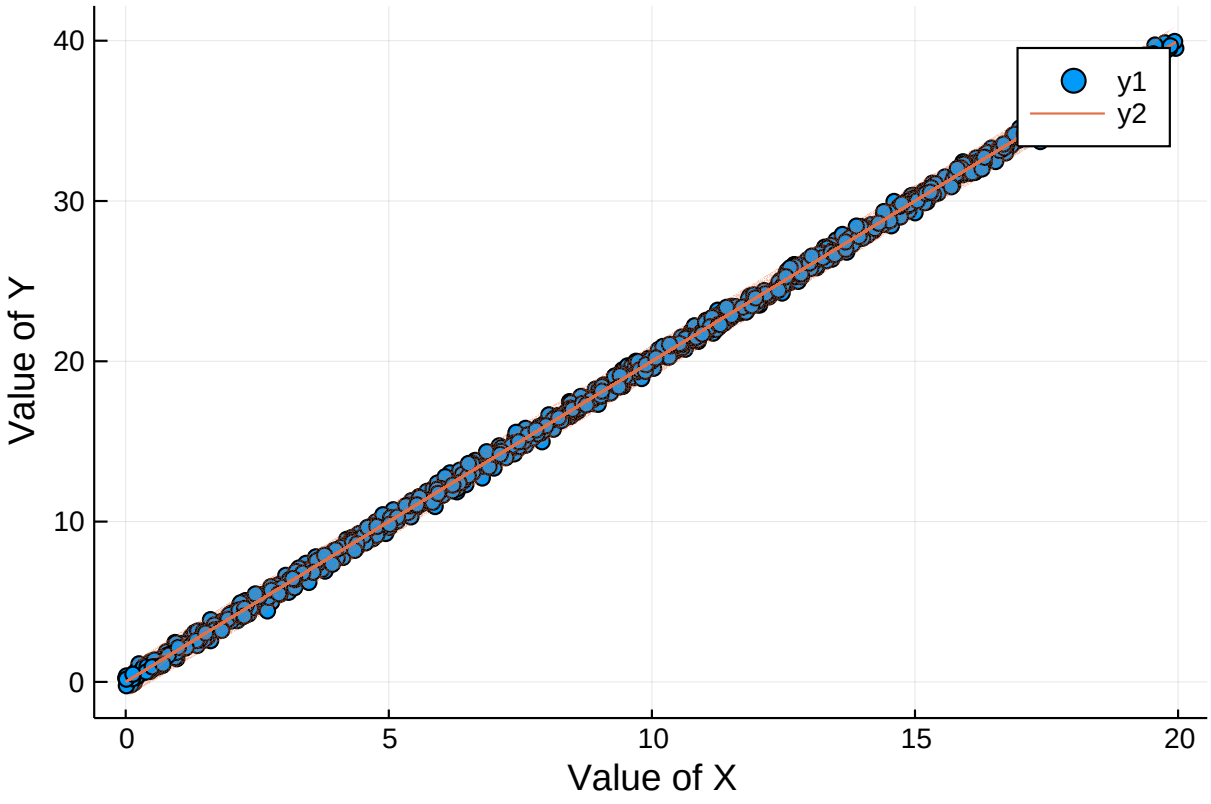
1. [2pts] Program the function that computes the the maximum likelihood estimate given X and Y. Use it to compute the estimate $\hat{\beta}$ for a $n = 1000$ sample from each target function.

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

n=1000 # batch_size


β_mle_1 = beta_mle(x1,y1)
println(β_mle_1)

[2.0001337134539097]

println("β_mle_1 is :",β_mle_1)

β_mle_1 is :[2.0001337134539097]

β_mle_2 = beta_mle(x2,y2)
println(β_mle_2)

[2.009615569205679]

println("β_mle_2 is :",β_mle_2)
```

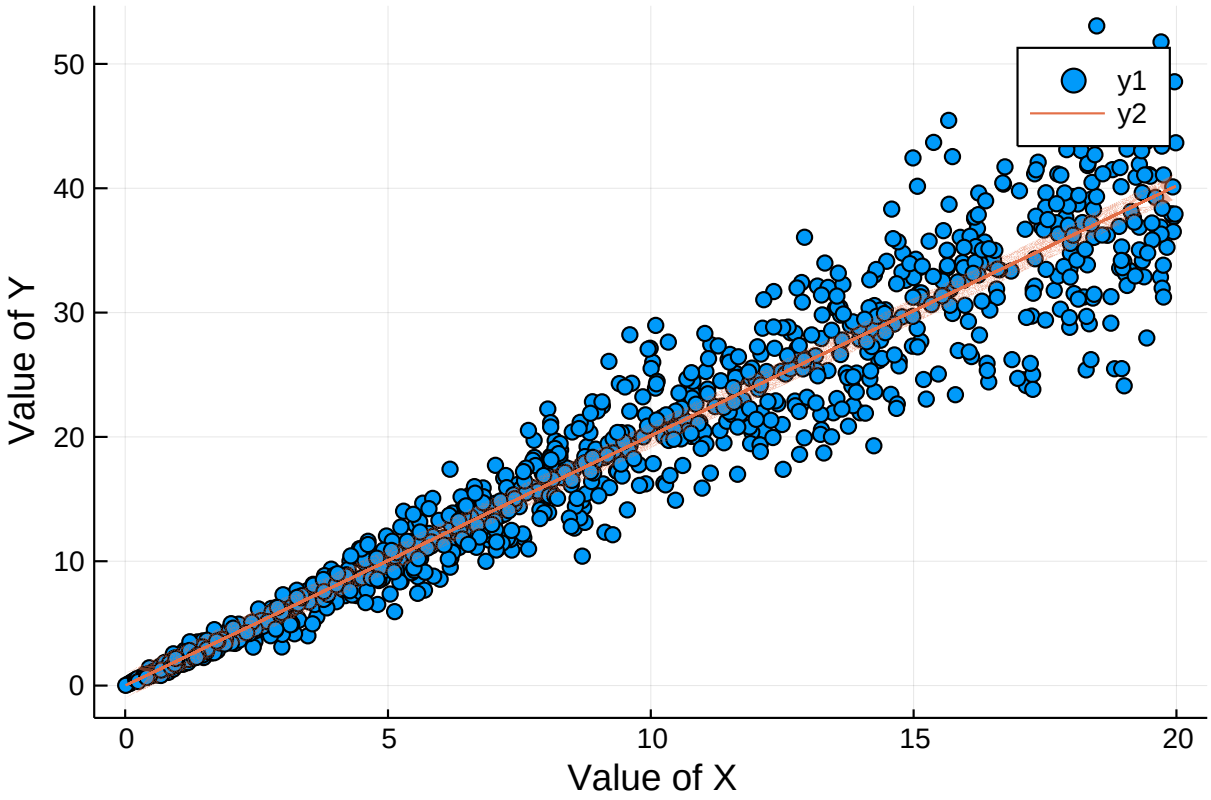
```
β_mle_2 is :[2.009615569205679]

β_mle_3 = beta_mle(x3,y3)
println("β_mle_3 is :",β_mle_3)

β_mle_3 is :[2.0632842627312336]
```

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**

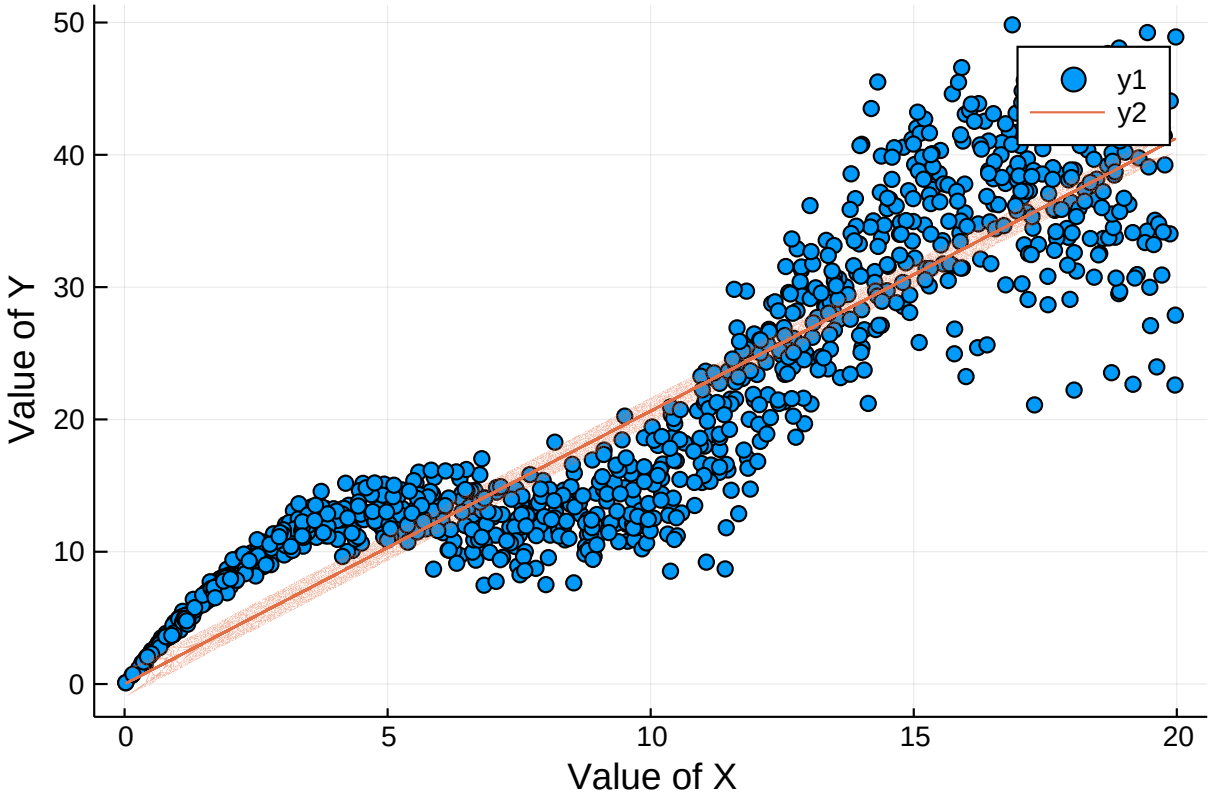
```
using Distributions
using Random
using  LinearAlgebra
reg1 = x1 .* β_mle_1
plot!(plot_f1,(x1)',(reg1)';ribbon=[-1,1])
```




```
using Distributions
using Random
using  LinearAlgebra
reg2 = x2 .* β_mle_2
plot!(plot_f2,(x2)',(reg2)';ribbon=[-1,1])
```




```
using Distributions
using Random
using  LinearAlgebra
reg3 = x3 .* β_mle_3
plot!(plot_f3,(x3)',(reg3)';ribbon=[-1,1])
```



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., $x_i \sim \mathcal{N}(\mu_i, \sigma_i)$.

```
using LinearAlgebra
function gaussian_log_likelihood(μ, σ, x)
  logL = -(1/2)*log(2*pi)-log.(σ)-(1/2)*inv(σ.^2)*(x-μ).^2
  return logL
end
```

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

```
using LinearAlgebra
function gaussian_log_likelihood(μ, σ, x)
  llx = -(1/2)*log(2*pi)-log.(σ)-(1/2)*inv(σ.^2)*(x-μ).^2
  return llx
end

# Test Gaussian likelihood against standard implementation
using Test
using Distributions: pdf, Normal
@testset "Gaussian log likelihood" begin
  # 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
end
```

```
Test Summary:           | Pass  Total
Gaussian log likelihood |    2      2
Test.DefaultTestSet("Gaussian log likelihood", Any[], 2, false)
```

```
using Test
using Distributions: pdf, Normal
@testset "Gaussian log likelihood" begin
  # 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
end
```

```
Test Summary:           | Pass  Total
Gaussian log likelihood |    2      2
Test.DefaultTestSet("Gaussian log likelihood", Any[], 2, false)
```

```
using Test
using Distributions: pdf, Normal
@testset "Gaussian log likelihood" begin
  # 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 |    2      2
Test.DefaultTestSet("Gaussian log likelihood", Any[], 2, 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.

```
using LinearAlgebra
function lr_model_nll(β,x,y;σ=1.)
  negll = -sum(gaussian_log_likelihood.(x'*β,σ,y))
  return negll
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: 65.56670451113682
--------  0.3  ------------
Negative Log-Likelihood: 2.7862942450269563
--------  1.0  ------------
Negative Log-Likelihood: 9.81669924838936
--------  2.0  ------------
Negative Log-Likelihood: 16.204224329815514
--------  target_f2  ------------
--------  0.1  ------------
Negative Log-Likelihood: 3590.044902136243
--------  0.3  ------------
Negative Log-Likelihood: 332.6892028031321
--------  1.0  ------------
```

```
Negative Log-Likelihood: 59.94895084222027
--------  2.0  ------------
Negative Log-Likelihood: 20.921004036072112
--------  target_f3  ------------
--------  0.1  ------------
Negative Log-Likelihood: 6922.737376789789
--------  0.3  ------------
Negative Log-Likelihood: 706.0122580112644
--------  1.0  ------------
Negative Log-Likelihood: 72.83462348922018
--------  2.0  ------------
Negative Log-Likelihood: 47.525382348758015
--------  100  ------------
--------  target_f1  ------------
--------  0.1  ------------
Negative Log-Likelihood: 332.19321792161867
--------  0.3  ------------
Negative Log-Likelihood: 19.054258669165012
--------  1.0  ------------
Negative Log-Likelihood: 95.67503600819025
--------  2.0  ------------
Negative Log-Likelihood: 162.28948441533186
--------  target_f2  ------------
--------  0.1  ------------
Negative Log-Likelihood: 66569.98248738747
--------  0.3  ------------
Negative Log-Likelihood: 7739.754725626423
--------  1.0  ------------
Negative Log-Likelihood: 534.3537079620546
--------  2.0  ------------
Negative Log-Likelihood: 305.4481708099968
--------  target_f3  ------------
--------  0.1  ------------
Negative Log-Likelihood: 139938.64130606002
--------  0.3  ------------
Negative Log-Likelihood: 13970.829533778624
--------  1.0  ------------
Negative Log-Likelihood: 1388.823299388103
--------  2.0  ------------
Negative Log-Likelihood: 450.75937900854444
--------  1000  ------------
--------  target_f1  ------------
--------  0.1  ------------
Negative Log-Likelihood: 3530.1769466320084
--------  0.3  ------------
Negative Log-Likelihood: 207.90873698579816
--------  1.0  ------------
Negative Log-Likelihood: 964.3658516272034
--------  2.0  ------------
Negative Log-Likelihood: 1623.7241306377778
--------  target_f2  ------------
--------  0.1  ------------
Negative Log-Likelihood: 624217.387862387
--------  0.3  ------------
Negative Log-Likelihood: 68891.2894019707
--------  1.0  ------------
Negative Log-Likelihood: 6941.380160725819
--------  2.0  ------------
Negative Log-Likelihood: 3054.3073365550176
```

```
--------  target_f3  ------------
--------  0.1  ------------
Negative Log-Likelihood: 1.2218053336689265e6
--------  0.3  ------------
Negative Log-Likelihood: 125286.06227546041
--------  1.0  ------------
Negative Log-Likelihood: 12751.124025764104
--------  2.0  ------------
Negative Log-Likelihood: 4338.124511790588
```

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

We compare the value of negative log likelihood among different values of σ for each target function under same sample size.

For target function 1, the best choice of σ is 0.3.

For target function 2, the best choice of σ is 2.0.

For the target function 3, the best choice of σ is 2.0.

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.

```
using Zygote: gradient
@testset "Gradients wrt parameter" begin
  β_test = randn()
  σ_test = rand()
  x,y = sample_batch(target_f1,100)
  ad_grad = gradient(β -> lr_model_nll(β,x,y;σ=σ_test),β_test)
  hand_derivative = -(1/σ_test^2)* ((x*y)-(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 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.

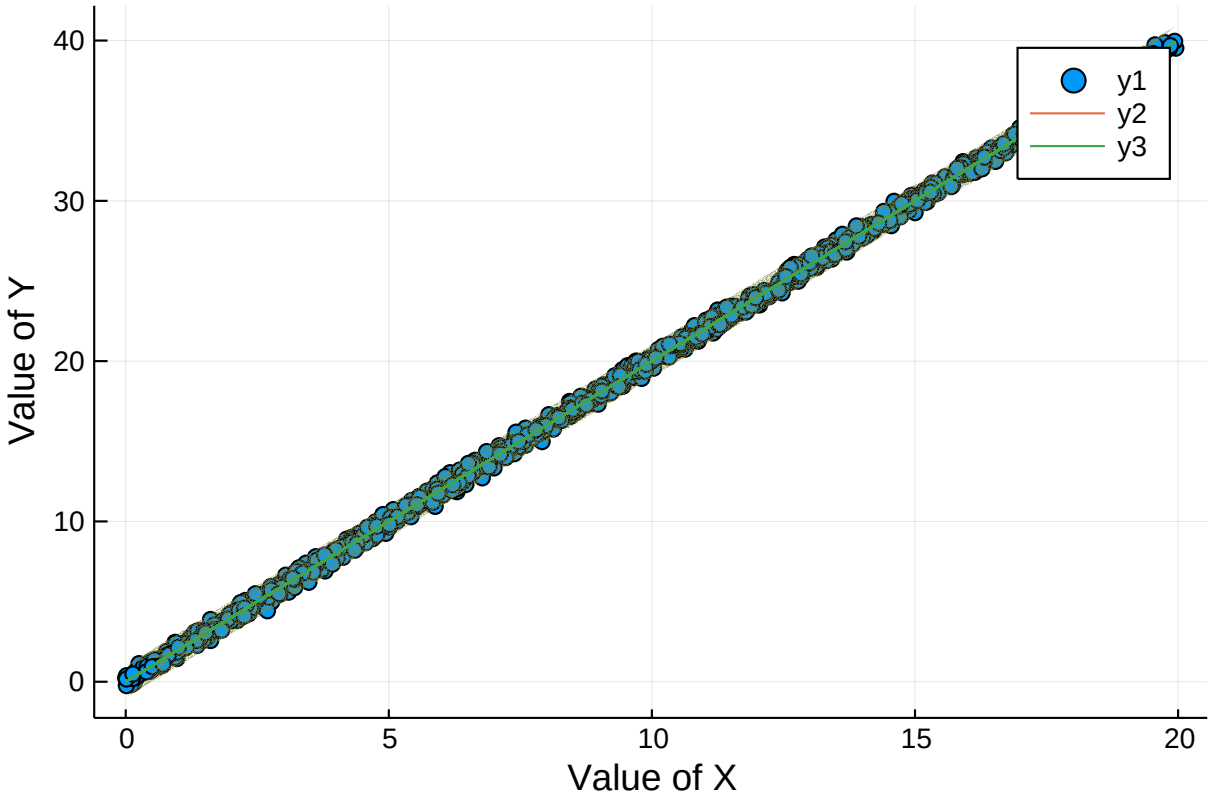
```
using Logging # Print training progress to REPL, not pdf
using Zygote: gradient
```

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

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)
      #@info "loss: $(lr_model_nll(β_curr,x,y;σ=σ_model))"
      grad_β = gradient(β -> lr_model_nll(β,x,y;σ=σ_model),β_curr)[1]
      β_curr = β_curr - grad_β*lr
    end
    return β_curr
end
```

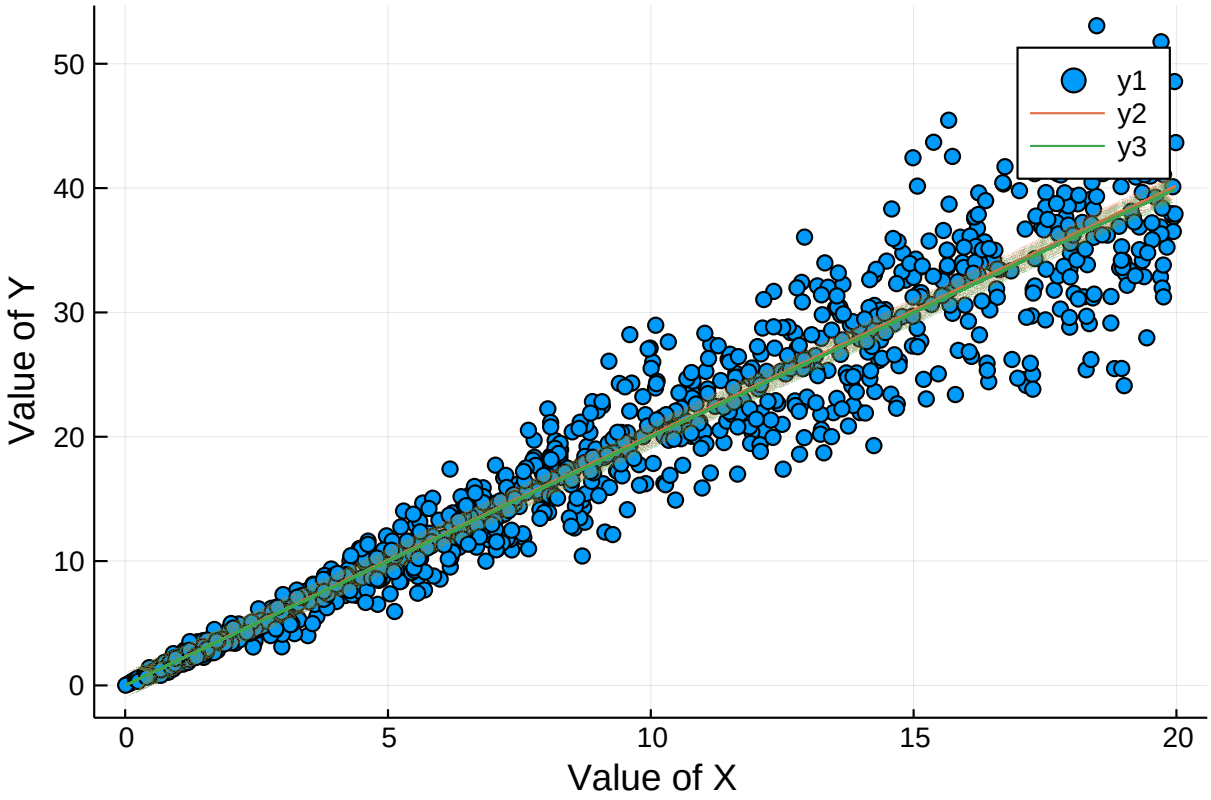
```
train_lin_reg (generic function with 1 method)
```

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.

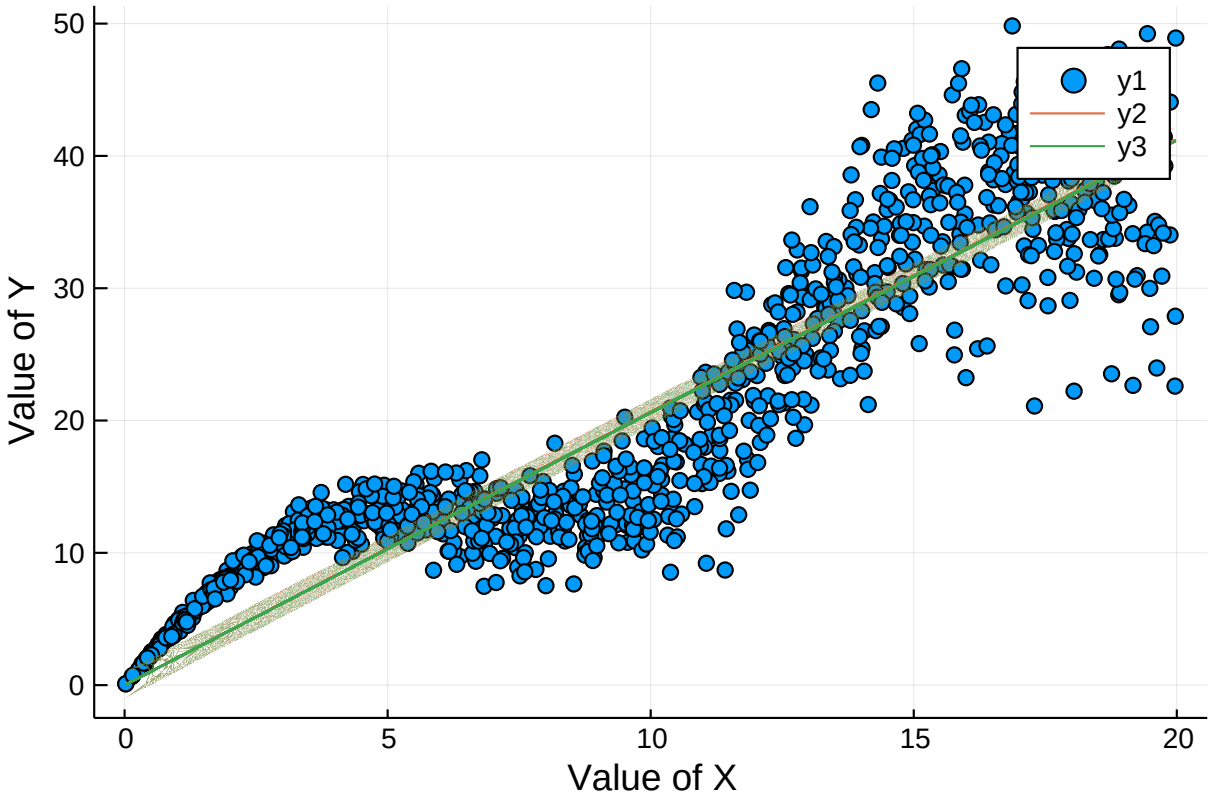
```
using Zygote: gradient
β_init = 1000*randn() # Initial parameter
β_learned_1= train_lin_reg(target_f1,β_init; bs= 1000, lr = 1e-6, iters=1000, σ_model =
1.)
lreg1 = x1 .* β_learned_1
plot!(plot_f1,(x1)',(lreg1)';ribbon=[-1,1])
```




```
using Zygote: gradient
β_init = 1000*randn() # Initial parameter
β_learned_2= train_lin_reg(target_f2,β_init; bs= 1000, lr = 1e-6, iters=1000, σ_model = 1.)
lreg2 = x2 .* β_learned_2
plot!(plot_f2,(x2)',(lreg2)';ribbon=[-1,1])
```



```
using Zygote: gradient
β_init = 1000*randn() # Initial parameter
β_learned_3= train_lin_reg(target_f3,β_init; bs= 1000, lr = 1e-6, iters=1000, σ_model =
1.)
lreg3 = x3 .* β_learned_3
plot!(plot_f3,(x3)',(lreg3)';ribbon=[-1,1])
```



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.

```
# Neural Network Function
function neural_net(x,θ)
  in = tanh.(x'*θ[1].+θ[2])
  out = in*θ[3].+θ[4]
  return vec(out)
end

# Random initial Parameters
inweight = rand(1,10)
inbias = rand(1,10)
outweight = rand(10,1)
outbias = rand(1,1)
θ = [inweight,inbias,outweight,outbias]

@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 $\sigma = 1.0$

```
using LinearAlgebra
function nn_model_nll(θ,x,y;σ=1)
  negll = -sum(gaussian_log_likelihood.(neural_net(x,θ),σ,y))
  return negll
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.

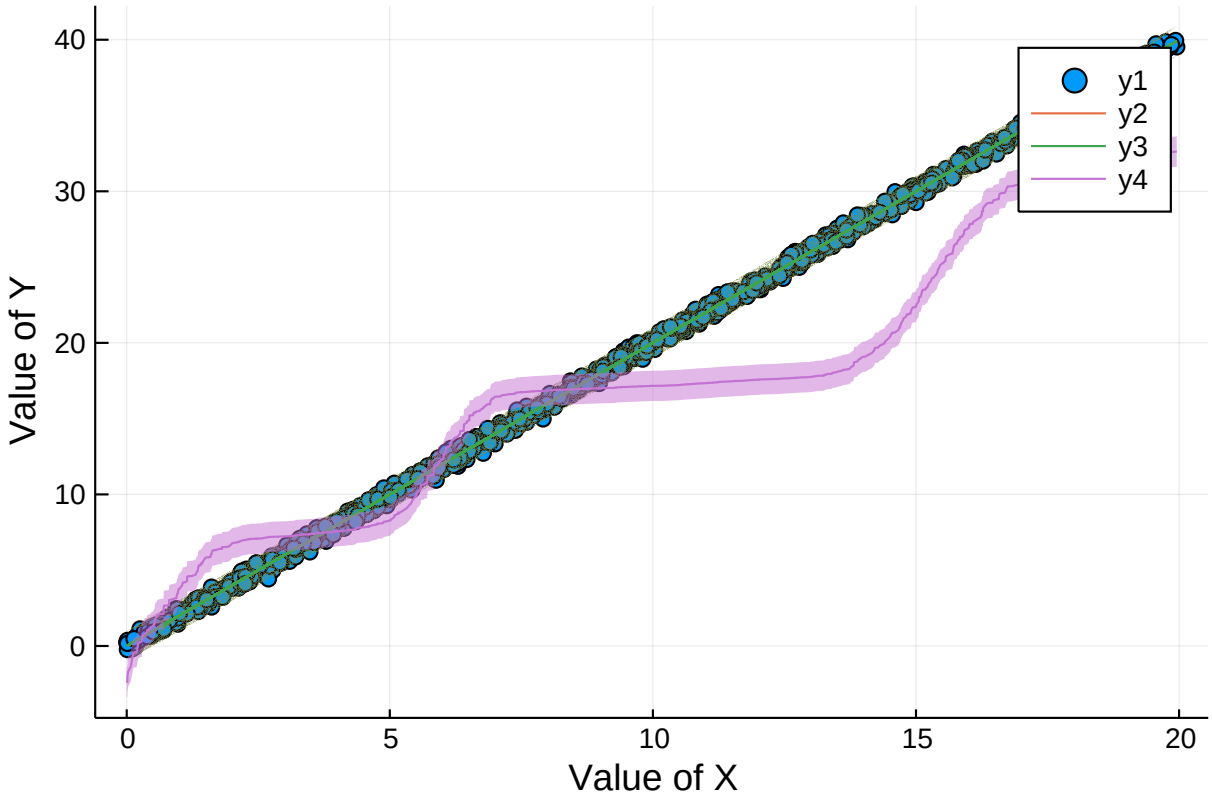
```
using Logging # Print training progress to REPL, not pdf

function train_nn_reg(target_f, θ_init; bs= 100, lr = 1e-5, iters=1000, σ_model = 1. )
    θ_curr = θ_init
```

```
    for i in 1:iters
      x,y = sample_batch(target_f,bs)
      #@info "loss: $(nn_model_nll(θ,x,y;σ=σ_model))"
      grad_θ = gradient(θ -> nn_model_nll(θ,x,y;σ=σ_model),θ_curr)[1]
      θ_curr = θ_curr - grad_θ*lr
    end
    return θ_curr
end
```

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

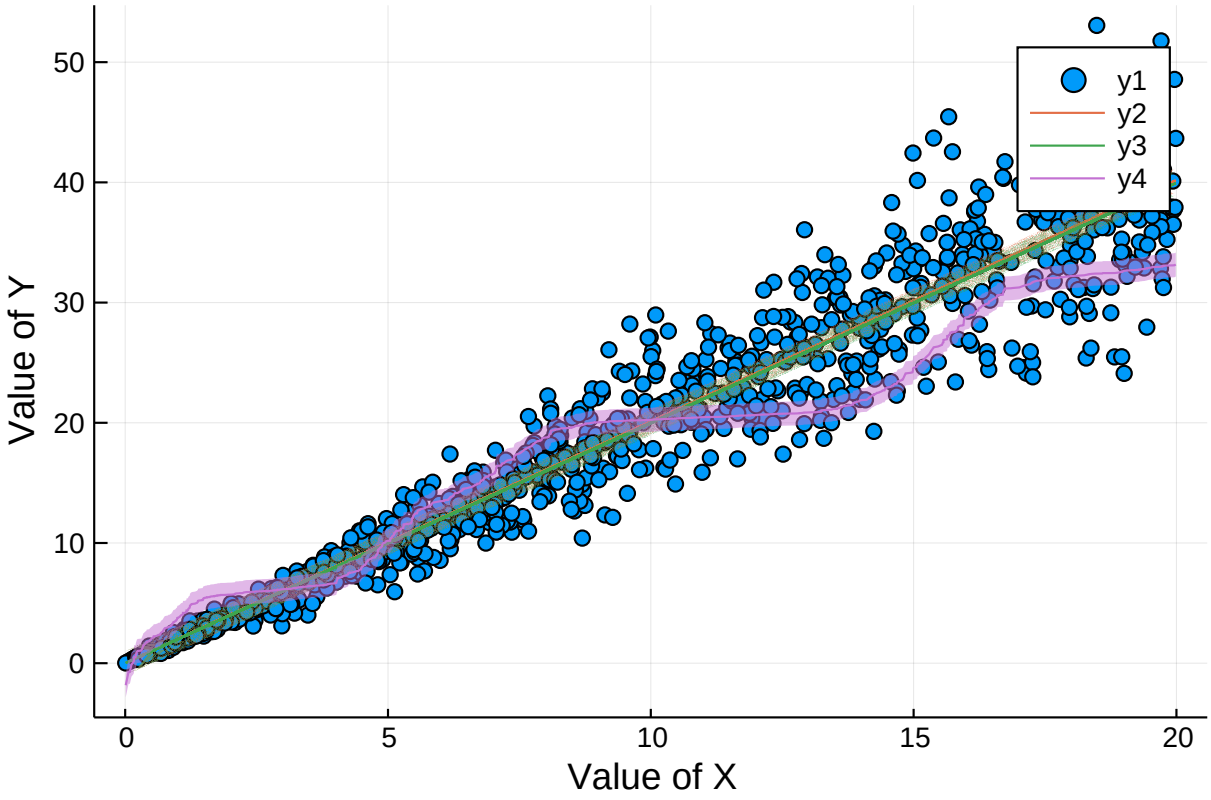
4. [2pts] For each target function, start with an initialization of the network parameters, θ, use your train function to minimize the negative log-likelihood and find an estimate for θ_{learned} by gradient descent. Then plot a $n = 1000$ sample of the data and the learned regression model with shaded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

```
using Zygote: gradient
inweight = rand(1,10)
inbias = rand(1,10)
outweight = rand(10,1)
outbias = rand(1,1)
θ_init = [inweight,inbias,outweight,outbias]
θ_learned_1 = train_nn_reg(target_f1, θ_init; bs= 1000, lr = 1e-5, iters=1000, σ_model =
1. )
nlreg1 = neural_net(x1,θ_learned_1)+rand(1000,1)
matrix = [x1' nlreg1]
newmatrix= sort(matrix,dims=1)
plot!(plot_f1,newmatrix[1:1000],newmatrix[1001:2000],ribbon=1)
```

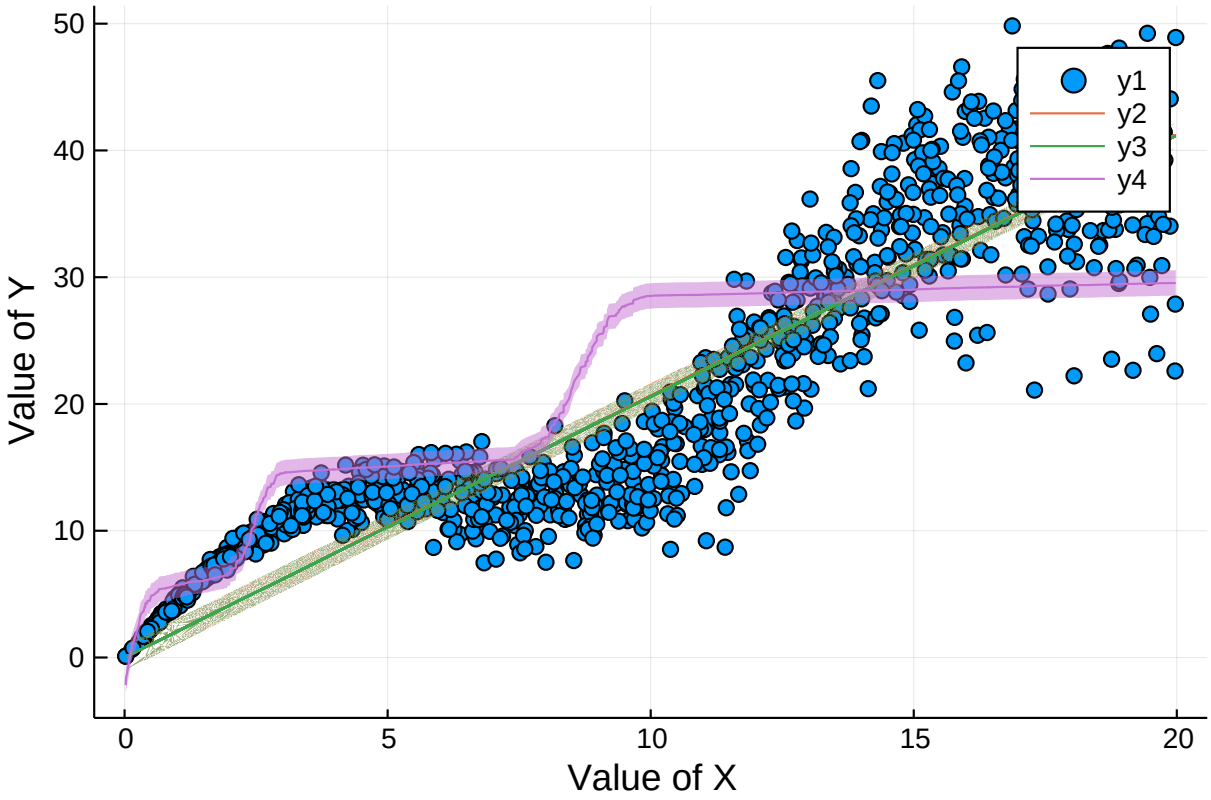



```
using Zygote: gradient
```

```
inweight = rand(1,10)
inbias = rand(1,10)
outweight = rand(10,1)
outbias = rand(1,1)
θ_init = [inweight,inbias,outweight,outbias]
θ_learned_2 = train_nn_reg(target_f2, θ_init; bs= 1000, lr = 1e-5, iters=1000, σ_model =
1. )
nlreg2 = neural_net(x2,θ_learned_2)+rand(1000,1)
matrix = [x2' nlreg2]
newmatrix= sort(matrix,dims=1)
plot!(plot_f2,newmatrix[1:1000],newmatrix[1001:2000],ribbon=1)
```




```
using Zygote: gradient
inweight = rand(1,10)
inbias = rand(1,10)
outweight = rand(10,1)
outbias = rand(1,1)
θ_init = [inweight,inbias,outweight,outbias]
θ_learned_3 = train_nn_reg(target_f3, θ_init; bs= 1000, lr = 1e-5, iters=1000, σ_model =
1. )
nlreg3 = neural_net(x3,θ_learned_3)+rand(1000,1)
matrix = [x3' nlreg3]
newmatrix= sort(matrix,dims=1)
plot!(plot_f3,newmatrix[1:1000],newmatrix[1001:2000],ribbon=1)
```



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.

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

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)
  in = tanh.(x'*θ[1].+θ[2])
  out1 = in*θ[3][1:10].+θ[4][1]
  out2 = in*θ[3][11:20].+θ[4][2]
  return out1,out2
end

# Random initial Parameters
inweight = rand(1,10)
inbias = rand(1,10)
outweight = rand(10,2)
outbias = rand(1,2)
θ = [inweight,inbias,outweight,outbias]

@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)
  negll =
-sum(gaussian_log_likelihood.(neural_net_w_var(x,θ)[1],exp.(neural_net_w_var(x,θ)[2]),y))
  return negll
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.

```
using Logging
function train_nn_w_var_reg(target_f, θ_init; bs= 100, lr = 1e-5, iters=1000)
    θ_curr = θ_init
    for i in 1:iters
      x,y = sample_batch(target_f,bs)
      #@info "loss: $(nn_with_var_model_nll(θ_curr,x,y))"
      grad_θ = gradient(θ -> nn_with_var_model_nll(θ,x,y),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)

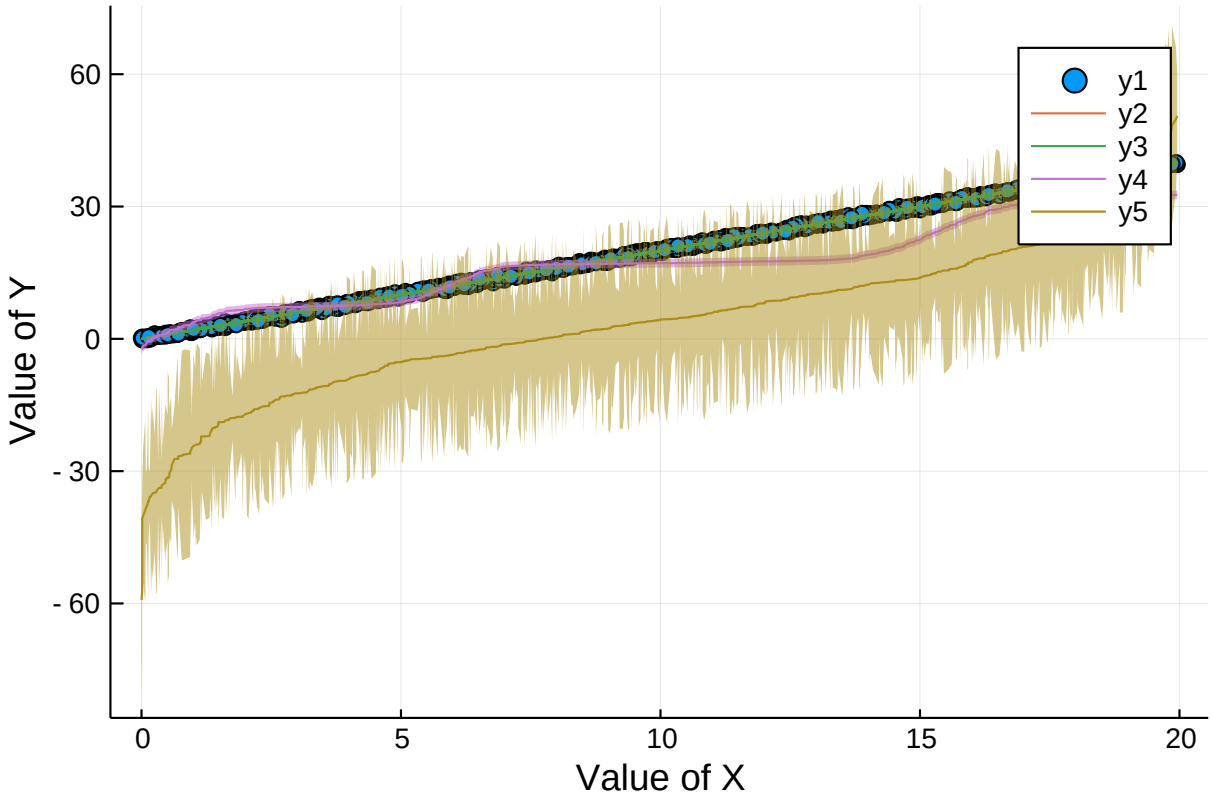
   Note: Learning the variance is tricky, and this may be unstable during training. There are some things you can try:

   - 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.

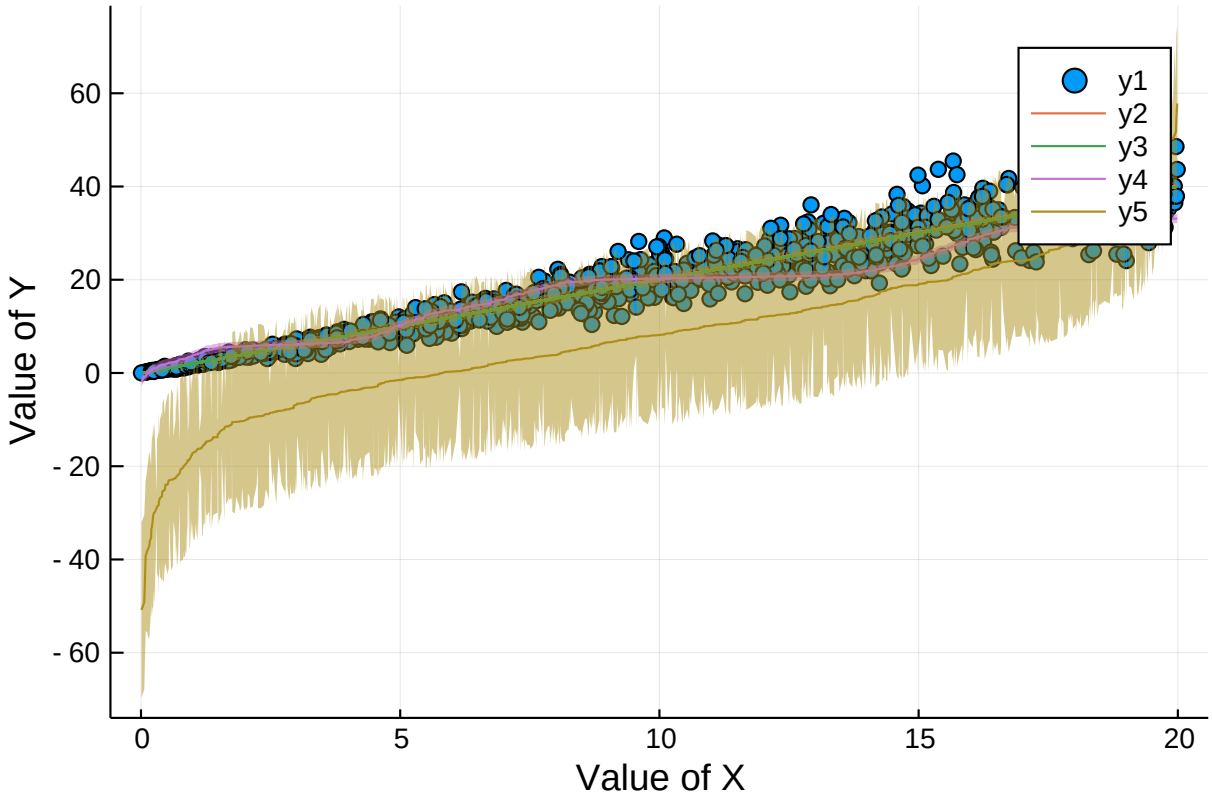
   For this question **you will not be assessed on the final quality of your model**. Specifically, if you fails to train an optimal model for the data that is okay. You are expected to learn something that is somewhat reasonable, and **demonstrates that this model is training and learning variance**.

   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.

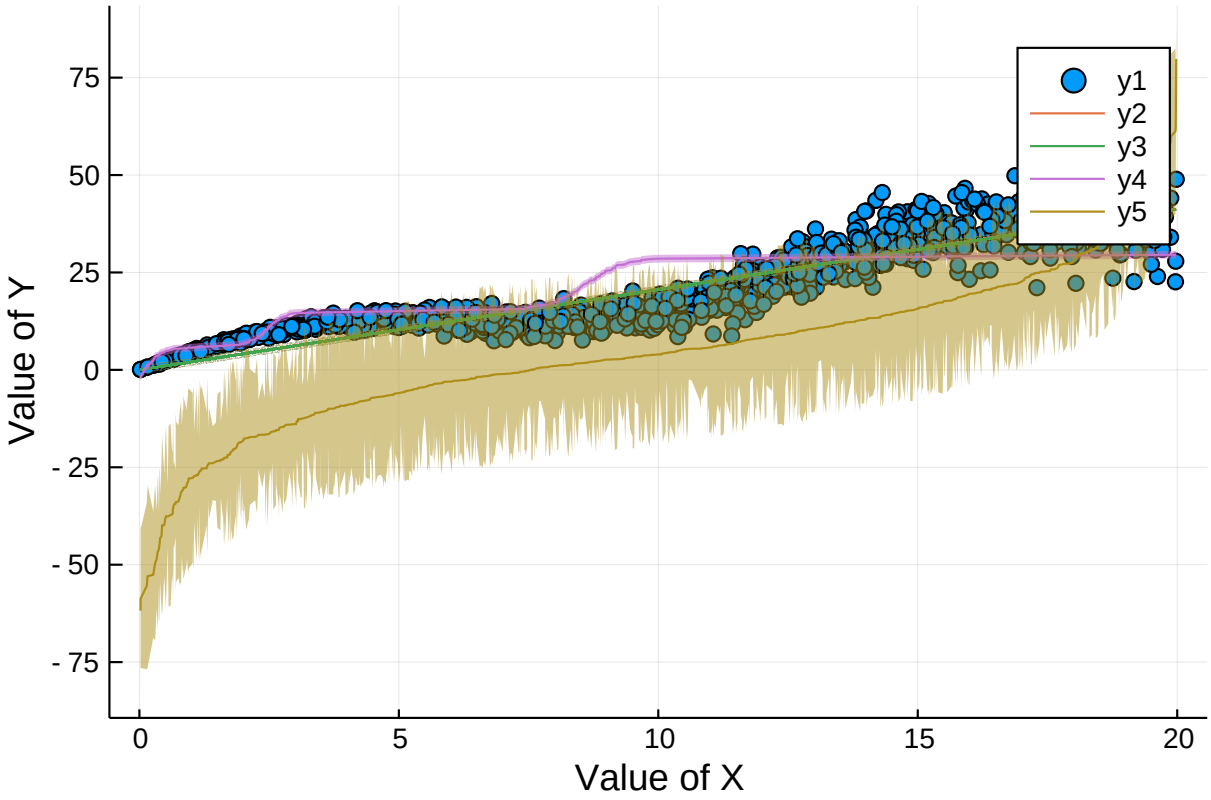
```
inweight = rand(1,10)
inbias = rand(1,10)
outweight = rand(10,2)
outbias = rand(1,2)
θ_init = [inweight,inbias,outweight,outbias]
θ_learned_1 = train_nn_w_var_reg(target_f1, θ_init)
nnlreg1 =
neural_net_w_var(x1,θ_learned_1)[1]+exp.(neural_net_w_var(x1,θ_learned_1)[2]).*randn(1000)
matrix = [x1' nnlreg1]
newmatrix= sort(matrix,dims=1)
plot!(plot_f1,newmatrix[1:1000],newmatrix[1001:2000],ribbon=exp.(neural_net_w_var(x1,θ_learned_1)[2]))
```




```
inweight = rand(1,10)
inbias = rand(1,10)
outweight = rand(10,2)
outbias = rand(1,2)
θ_init = [inweight,inbias,outweight,outbias]
θ_learned_2 = train_nn_w_var_reg(target_f2, θ_init)
nnlreg2 =
neural_net_w_var(x2,θ_learned_2)[1]+exp.(neural_net_w_var(x2,θ_learned_2)[2]).*randn(1000)
matrix = [x2' nnlreg2]
newmatrix= sort(matrix,dims=1)
plot!(plot_f2,newmatrix[1:1000],newmatrix[1001:2000],ribbon=ribbon=exp.(neural_net_w_var(x2,θ_learned_
```




```
inweight = rand(1,10)
inbias = rand(1,10)
outweight = rand(10,2)
outbias = rand(1,2)
θ_init = [inweight,inbias,outweight,outbias]
θ_learned_3 = train_nn_w_var_reg(target_f3, θ_init)
nnlreg3 =
neural_net_w_var(x3,θ_learned_3)[1]+exp.(neural_net_w_var(x3,θ_learned_3)[2]).*randn(1000)
matrix = [x3' nnlreg3]
newmatrix= sort(matrix,dims=1)
plot!(plot_f3,newmatrix[1:1000],newmatrix[1001:2000],ribbon=exp.(neural_net_w_var(x3,θ_learned_3)[2]))
```



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.