Assignment 1

Raj Patel

February 25, 2020

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

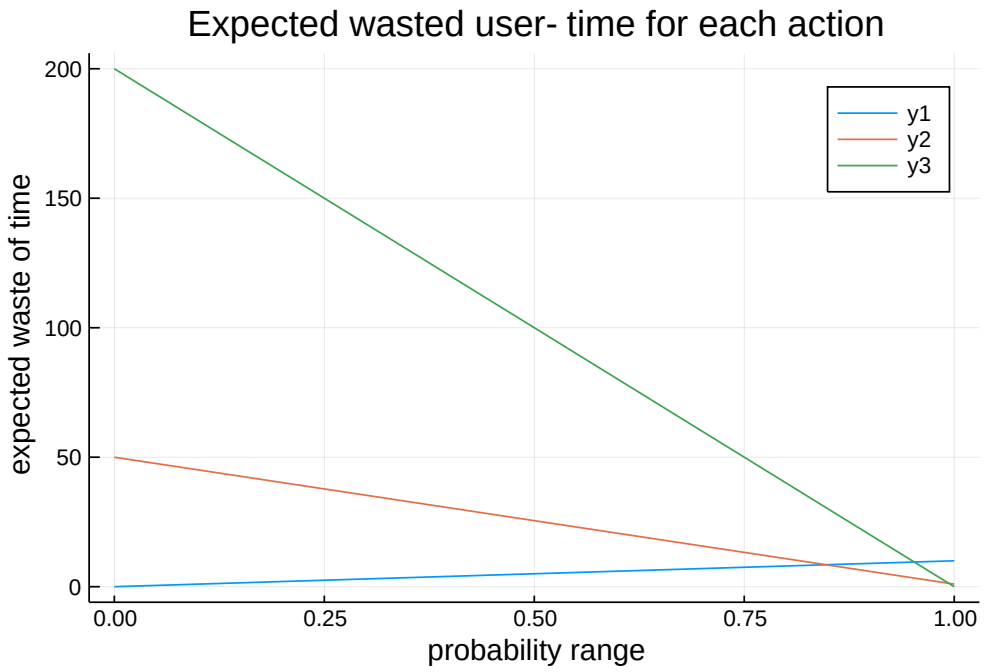
Depending on whether or not the email really is spam, the user will suffer a different amount of wasted time for the different actions we can take, $L(\text{action}, \text{spam})$:

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

1. [3pts] Plotting 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)

    # Action 1: "Showing email to the user"
    # Action 2: "Putting email in spam folder"
    # Action 3: "Deleting email entirely"

    convert(AbstractFloat, (losses[action,][1] .* prob_spam) + (losses[action,][2] .* (1
- prob_spam)))

end
```

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

# Make plot
using Plots

# y1 = show, y2 = spam, y3 = delete
for action in 1:num_actions
  display(plot!(prob_range, expected_loss_of_action.(prob_range, action),title="Expected
wasted user-time for each action",xlabel="probability range",
  ylabel="expected waste of time"))
end
```



2. [2pts] Function that computes the optimal action given the probability of spam.

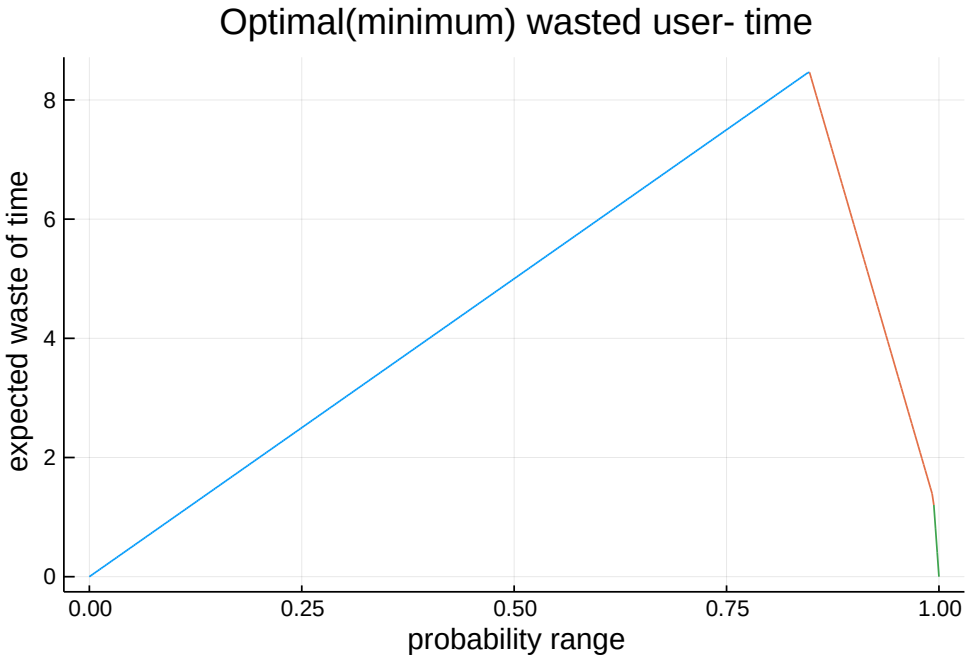
```
function optimal_action(prob_spam)

      # Creating a vector to store wasted time for each action
      wasted_time = zeros(num_actions)

      # Finding wasted time for each action
      for action in 1:num_actions
        wasted_time[action] = (expected_loss_of_action(prob_spam, action))
      end

      # Finding the index of action for which minimum time is wasted (i.e., best
action)
      findmin(wasted_time)[2]
end
```

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

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

Coloring 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

    # Finding optimal actions for all values of p and appending those in the
"optimal_actions" vector
    append!(optimal_actions, optimal_action(p))

    # Finding optimal losses for all values of p and appending them in the
"optimal_actions" vector
    append!(optimal_losses, expected_loss_of_action(p,optimal_action(p)))

end

# Plotting prob_range vs optimal_losses with line colour indicating which action
should be taken

# line colour: blue = action 1(show email), orange = action 2(put in spam), green =
action 3(delete)
plot(prob_range, optimal_losses, linecolor=optimal_actions, legend = false,
title="Optimal(minimum) wasted user-time", xlabel="probability range", ylabel="expected
waste of time")
```



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.

Answer:

Here, we want to find range of probabilities of an email being spam for which we could delete it. This is equivalent to finding range of probabilities of an email being spam where the expected waster user time is minimum if we delete it.

So, for that we have two equations and we want to find p(spam—email) which satisfies both the equations: (Here, we call $x = p(spam|email)$)

$$(0 \times x) + (200 \times (1-x)) \leq (1 \times x) + (50 \times (1-x)) \quad (1)$$
$$(0 \times x) + (200 \times (1-x)) \leq (10 \times x) + (0 \times (1-x)) \quad (2)$$

$$(0x) + (200(1-x)) \leq (1x) + (50(1-x))$$
$$200 - 200x \leq x + 50 - 50x$$
$$150 \leq 151x$$
$$\boxed{x \geq \frac{150}{151}}$$

$$(0x) + (200(1-x)) \leq (10x) + (0(1-x))$$
$$200 - 200x \leq 10x$$
$$200 \leq 210x$$
$$\boxed{x \geq \frac{200}{210}}$$

Only for $\boxed{x \geq \frac{150}{151}}$ will the above two equations be satisfied. So, for $x \in [\frac{150}{151}, 1]$, where x = p(spam given email), 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 $n < m$?

Answer:

If we consider the simplest case with m=2 and n=1, we are trying to fit a line to 1 point. As a result, we will be able to fit infinite number of lines and not get a unique solution.

From a linear algebra perspective, if we try to estimate $\hat{\beta}$, we know that it is a solution of a system of n equations with m unknowns. So, there will be no unique solution to this system where m > n.

(Though not necessary) But if we look from a more rigorous perspective, if n < m, we are left with an under-determined system of equations. To solve that, we can use pseudo-inverse as that helps us find the smallest solution which happens to be the one with lowest variance.

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

Answer:

We know that $Y \sim \mathcal{N}(X^T\beta, \sigma^2 I)$. Using that, we have:

$$\begin{aligned}\mathbb{E}[(XX^T)^{-1}XY] &= (XX^T)^{-1}X\mathbb{E}(Y) \\ &= (XX^T)^{-1}XX^T\beta \\ &= \boxed{\beta}\end{aligned}$$

$$\begin{aligned}Cov[(XX^T)^{-1}XY] &= ((XX^T)^{-1}X)Var(Y)((XX^T)^{-1}X)^T \\ &= ((XX^T)^{-1}X)(\sigma^2 I)((XX^T)^{-1}X)^T \\ &= \sigma^2((XX^T)^{-1}XX^T(XX^T)^{-1}) \\ &= \boxed{\sigma^2(XX^T)^{-1}}\end{aligned}$$

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

Answer:

- Likelihood is $\prod_{i=1}^{n} f(y_i|\beta)$. From this, we can say that the likelihood we try to maximize is $\frac{1}{(2\pi\sigma^2)^{n/2}} exp(\frac{-1}{2\sigma^2}(Y - X^T\beta)^2)$ as $\prod_{i=1}^{n} f(y_i|\beta) = \frac{1}{(2\pi\sigma^2)^{n/2}} exp(\frac{-1}{2\sigma^2}(Y - X^T\beta)^2)$
- Now, as exponential is monotonic and $(2\pi\sigma^2)^{n/2} > 0$,we can say that maximizing the likelihood is same as maximizing $\frac{-1}{2\sigma^2}(Y - X^T\beta)^2$.
- Using this, we can say that maximizing $\frac{-1}{2\sigma^2}(Y - X^T\beta)^2$ is same as minimizing $(Y - X^T\beta)^2$ as $2\sigma^2 > 0$ and there is a "negative" sign in the function we are maximizing.
- Finally, we know that $(Y - X^T\beta)^2$ is squared error.
- So, starting with likelihood, we have shown that maximizing likelihood is equivalent to minimizing the squared error.

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

Answer:

Here, we want to show $\sum_{i=1}^{n}(y_i - x_i\beta)^2$ in vector notation. The squared error in vector terms can be written as:

$$\begin{aligned}(Y - X^T\beta)^T(Y - X^T\beta) &= (Y^T - \beta^T X)(Y - X^T\beta) \\ &= Y^T Y - Y^T X^T\beta - \beta^T XY + \beta^T XX^T\beta \\ &= \boxed{Y^T Y - 2Y^T X^T\beta + \beta^T XX^T\beta} (\because Y^T X^T\beta = \beta^T XY)\end{aligned}$$

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.

Answer:

We know that our likelihood is $\prod_{i=1}^{n} f(y_i|\beta)$.

Moreover,from part (3), $\prod_{i=1}^{n} f(y_i|\beta) = \frac{1}{(2\pi\sigma^2)^{n/2}} exp(\frac{-1}{2\sigma^2}(Y - X^T\beta)^2)$

Therefore, our log-likelihood is, $l = log(\prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} exp(\frac{-1}{2\sigma^2}(Y - X^T\beta)^2))$

So, our negative log-likelihood is, $nll = \frac{n}{2} log(2\pi\sigma^2) + \frac{1}{2\sigma^2}(Y - X^T\beta)^T(Y - X^T\beta)$

Taking derivative of negative log-likelihood with respect to β, we have, $\frac{\partial(\frac{n}{2}log(2\pi\sigma^2) + \frac{1}{2\sigma^2}(Y-X^T\beta)^T(Y-X^T\beta))}{\partial\beta} = \frac{-1}{2\sigma^2} 2X(Y - X^T\beta)$

Now, setting this to 0, we get, $\frac{-1}{2\sigma^2} 2X(Y - X^T\beta) = 0$

Therefore, $XY = XX^T\beta$

So, $\beta = \boxed{(XX^T)^{-1}XY}$

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] Function which produces a batch of data $x \sim \text{Uniform}(0, 20)$ and `y = target_f(x)`

```
using Distributions

function sample_batch(target_f, batch_size)

  # rand() produces number uniformly between 0 and 1.
  # To generate numbers uniformly in [a,b], "rand()*(b-a) + a" should work
  x = (rand(1,batch_size) * 20) .+ 0

  y = target_f(x)
  return(x,y)
end
```

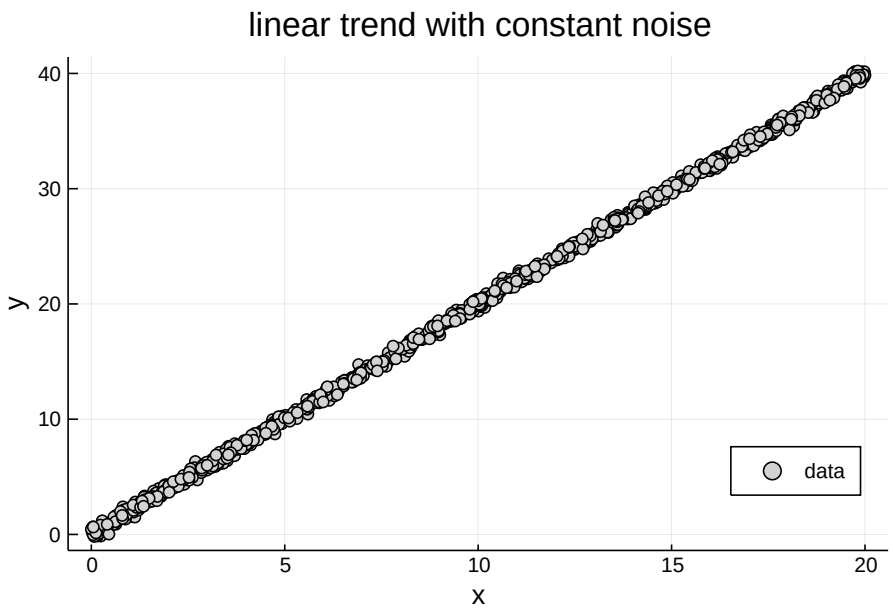
```
sample_batch (generic function with 1 method)
```

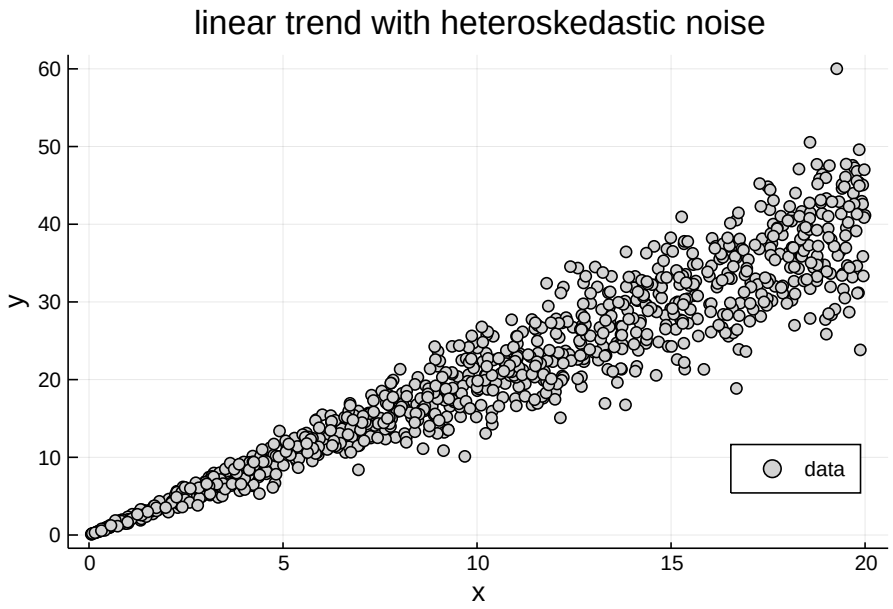
```
using Test
@testset "sample dimensions are correct" begin
  m = 1 # dimensionality
  n = 200 # batch-size
  for target_f in (target_f1, target_f2, target_f3)
    x,y = sample_batch(target_f,n)
    @test size(x) == (m,n)
    @test size(y) == (n,)
  end
end
```

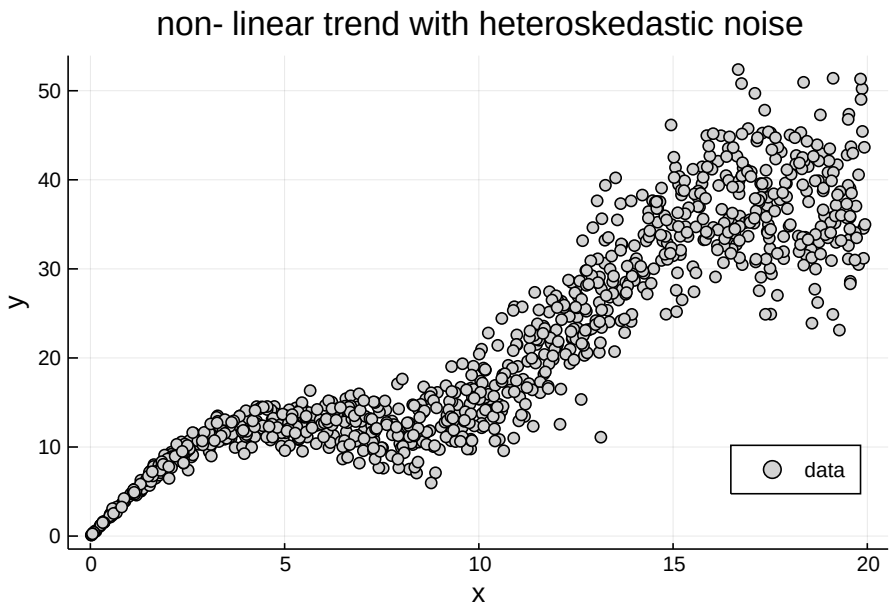
```
Test Summary:                  | Pass  Total
sample dimensions are correct |    6      6
Test.DefaultTestSet("sample dimensions are correct", Any[], 6, false)
```

2. [1pts] For all three targets, we will plot a $n = 1000$ sample of the data.

```
using Plots

x1,y1 = sample_batch(target_f1,1000)
plot_f1_scatter = plot(x1[1,:], y1, seriestype=:scatter, color = [:lightgray], legend =
:bottomright,title="linear trend with constant noise", xlabel="x", ylabel="y")

x2,y2 = sample_batch(target_f2,1000)
plot_f2_scatter = plot(x2[1,:], y2, seriestype=:scatter, color = [:lightgray], legend =
:bottomright,title="linear trend with heteroskedastic noise", xlabel="x", ylabel="y")


x3,y3 = sample_batch(target_f3,1000)
plot_f3_scatter = plot(x3[1,:],y3, seriestype=:scatter, color = [:lightgray], legend =
:bottomright,title="non-linear trend with heteroskedastic noise", xlabel="x", ylabel="y")
```


linear trend with constant noise
y
x
data
0
10
20
30
40
0
5
10
15
20


linear trend with heteroskedastic noise
y
x
data
0
10
20
30
40
50
60
0
5
10
15
20


non- linear trend with heteroskedastic noise
y
x
data
0
10
20
30
40
50
0
5
10
15
20

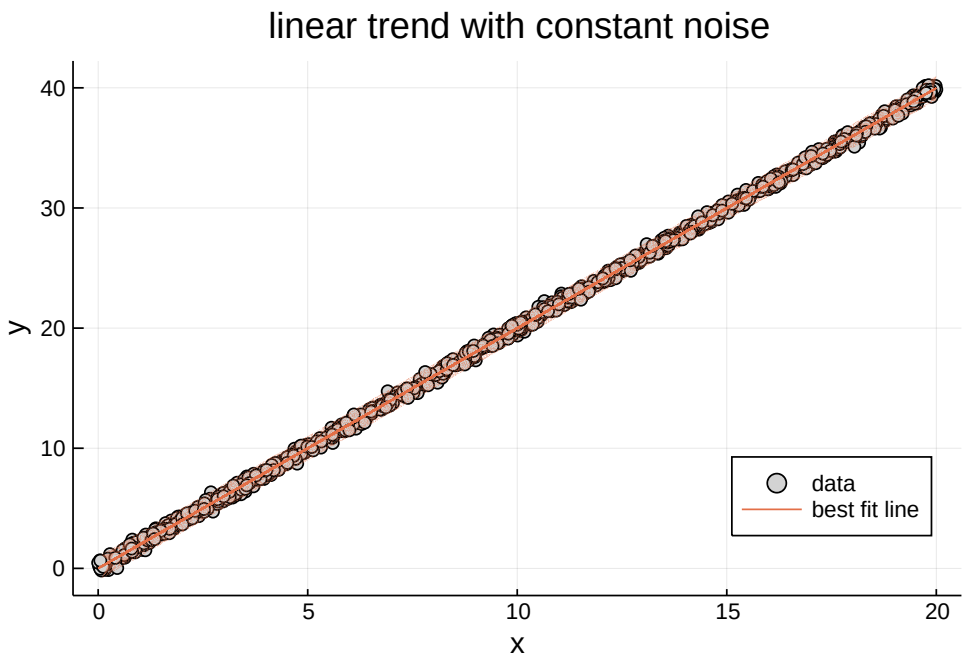
2.3 Linear Regression Model with $\hat{\beta}$ MLE [4pts]

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.

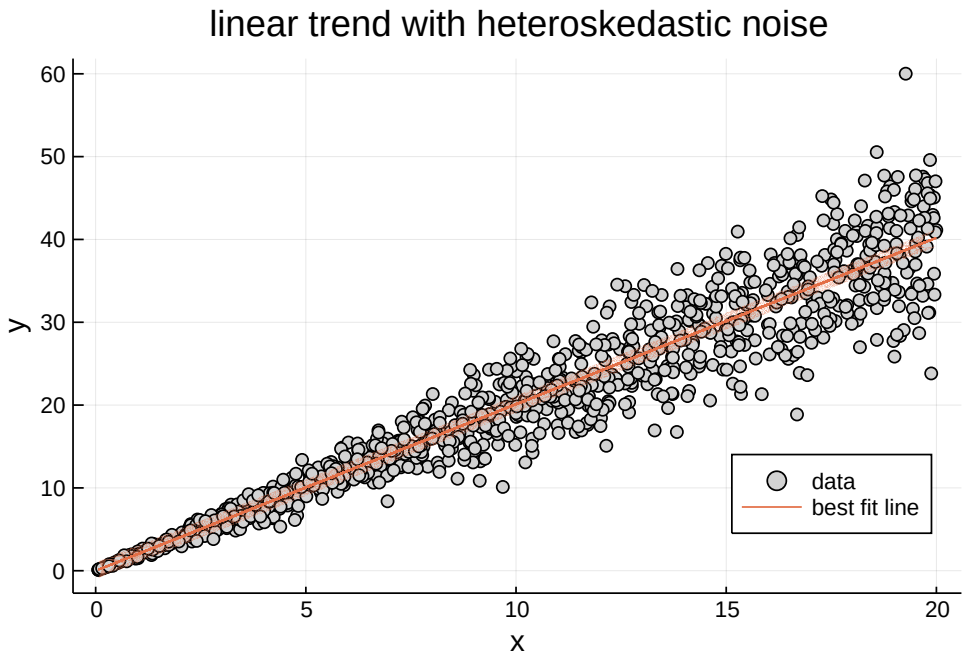
```
function beta_mle(X,Y)
  # Based on the formula derived in 2.1.5, beta_mle will be as follows:
  beta = inv(X*X')*(X*Y)
  return beta
end

n=1000 # batch_size

# Estimating beta_mle for n=1000 sample generated previously from each target
function

β_mle_1 = beta_mle(x1,y1)

β_mle_2 = beta_mle(x2,y2)

β_mle_3 = beta_mle(x3,y3)
```

2. [2pts] For each function, we plot the linear regression model given by $Y \sim \mathcal{N}(X^T\hat{\beta}, \sigma^2 I)$ for $\sigma = 1.$. This plot will 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. **We display 3 plots, one for each target function, showing samples of data and maximum likelihood estimate linear regression model**

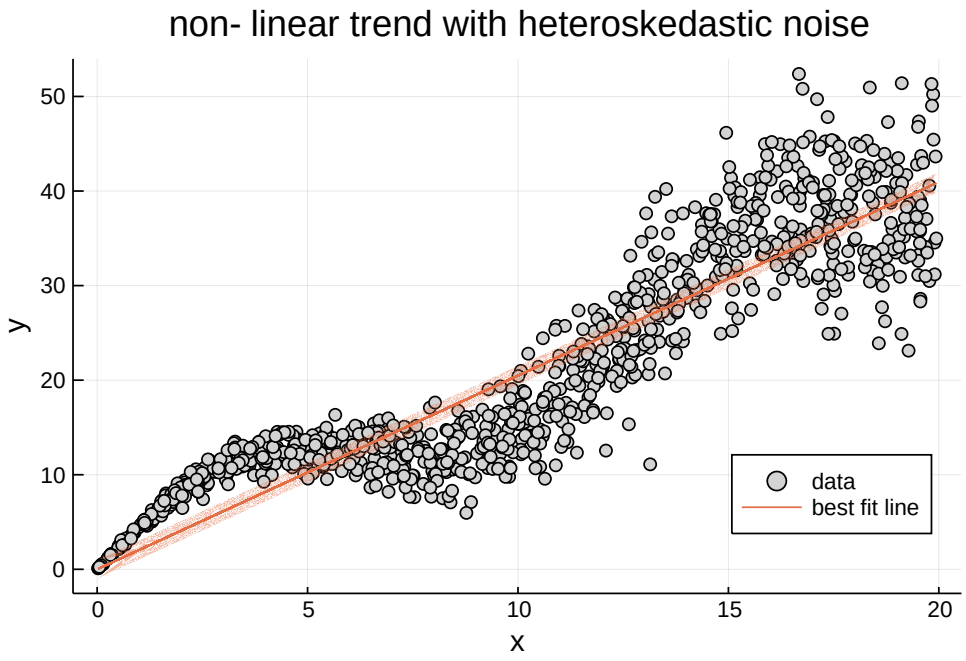
```
# Linear trend with constant noise
plot!(plot_f1_scatter, x1[1,:],(β_mle_1 .* x1)[1,:],ribbon=[-1,1])
```



```
# Linear trend with heteroskedastic noise
plot!(plot_f2_scatter, x2[1,:],(β_mle_2 .* x2)[1,:],ribbon=[-1,1])
```




```
# Non-linear trend with heteroskedastic noise
plot!(plot_f3_scatter, x3[1,:],(β_mle_3 .* x3)[1,:],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., $x_i \sim \mathcal{N}(\mu_i, \sigma_i)$.

```
function gaussian_log_likelihood(μ, σ, x)
  # Calculating log likelihood based on the derived results from Q2
  ll = ((-length(x) / 2) * (log(2 * pi))) .- ((length(x) / 2) * (log(σ .^ 2))) .- ((1/(2
.* (σ .^ 2))) * (sum(x .- μ)^2))
  return ll
end
```

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

```
using Test
@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.

```
function lr_model_nll(β,x,y;σ=1.)
  return -sum(gaussian_log_likelihood.(x'.* β, σ, y))
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: 20.392345422129136
--------  0.3  ------------
Negative Log-Likelihood: 4.132078023918309
--------  1.0  ------------
Negative Log-Likelihood: 9.884020335138928
--------  2.0  ------------
Negative Log-Likelihood: 16.189659974160534
--------  target_f2  ------------
--------  0.1  ------------
Negative Log-Likelihood: 12047.238561526765
--------  0.3  ------------
Negative Log-Likelihood: 656.8295349248251
--------  1.0  ------------
Negative Log-Likelihood: 36.27505884793924
--------  2.0  ------------
Negative Log-Likelihood: 24.771130195230477
--------  target_f3  ------------
--------  0.1  ------------
Negative Log-Likelihood: 24835.415202474433
--------  0.3  ------------
Negative Log-Likelihood: 882.1815066245987
--------  1.0  ------------
Negative Log-Likelihood: 147.02124454315575
--------  2.0  ------------
Negative Log-Likelihood: 47.23970000572146
--------  100  ------------
--------  target_f1  ------------
--------  0.1  ------------
Negative Log-Likelihood: 344.4721114332799
--------  0.3  ------------
Negative Log-Likelihood: 32.36146382848095
--------  1.0  ------------
Negative Log-Likelihood: 97.21610299844836
--------  2.0  ------------
Negative Log-Likelihood: 162.1609261949322
--------  target_f2  ------------
--------  0.1  ------------
Negative Log-Likelihood: 39457.06156755608
--------  0.3  ------------
Negative Log-Likelihood: 4923.448133515954
--------  1.0  ------------
Negative Log-Likelihood: 660.0593913252882
--------  2.0  ------------
Negative Log-Likelihood: 287.07668566624585
--------  target_f3  ------------
--------  0.1  ------------
Negative Log-Likelihood: 104430.47012846144
--------  0.3  ------------
```

```
Negative Log-Likelihood: 15298.686338930163
--------  1.0  ------------
Negative Log-Likelihood: 1327.119819699227
--------  2.0  ------------
Negative Log-Likelihood: 419.7642131873153
--------  1000  ------------
--------  target_f1  ------------
--------  0.1  ------------
Negative Log-Likelihood: 2920.616411982549
--------  0.3  ------------
Negative Log-Likelihood: 209.23126651972055
--------  1.0  ------------
Negative Log-Likelihood: 967.2864450234912
--------  2.0  ------------
Negative Log-Likelihood: 1623.7133463198022
--------  target_f2  ------------
--------  0.1  ------------
Negative Log-Likelihood: 600986.688113208
--------  0.3  ------------
Negative Log-Likelihood: 73179.83543483635
--------  1.0  ------------
Negative Log-Likelihood: 6891.079687105353
--------  2.0  ------------
Negative Log-Likelihood: 3036.2514796249097
--------  target_f3  ------------
--------  0.1  ------------
Negative Log-Likelihood: 1.1134643569930636e6
--------  0.3  ------------
Negative Log-Likelihood: 123453.19064070823
--------  1.0  ------------
Negative Log-Likelihood: 13197.071802501188
--------  2.0  ------------
Negative Log-Likelihood: 4432.698953613653
```

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

Here, σ and batch-size n are modelling hyperparameters. In the expression of maximum likelihood estimate, σ or n do not appear, and therefor in principle shouldn't affect the final answer. However, in practice these can have significant effect on the numerical stability of the model. Too small values of σ will make data away from the mean very unlikely, which can cause issues with precision. Also, the negative log-likelihood objective involves a sum over the log-likelihoods of each datapoint. This means that with a larger batch-size n, there are more datapoints to sum over, so a larger negative log-likelihood is not necessarily worse. So, we cannot directly compare the negative log-likelihoods achieved by these models with different hyperparameter settings.

In our case, for target function 1, we observe that if we consider smaller negative log-likelihood as the deciding factor, $\sigma = 0.3$ gives the best result. For target function 2, $\sigma = 2.0$ gives the best result. However, for target function 3, there is no single value of σ which gives smallest negative log likelihood for n=10,100 and 1000.

2.5 Automatic Differentiation and Maximizing Likelihood [3pts]

In a previous question we 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, we now use automatic differentiation to compute the derivative of the negative log-likelihood of the sampled data with respect β. Then, we test that this is equivalent to the hand-derived value.

```
using Zygote: gradient
using Test
@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 = (-(x / (σ_test^2))) * (y - (x' *  β_test))
    @test ad_grad[1] ≈ hand_derivative[1]
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] We 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

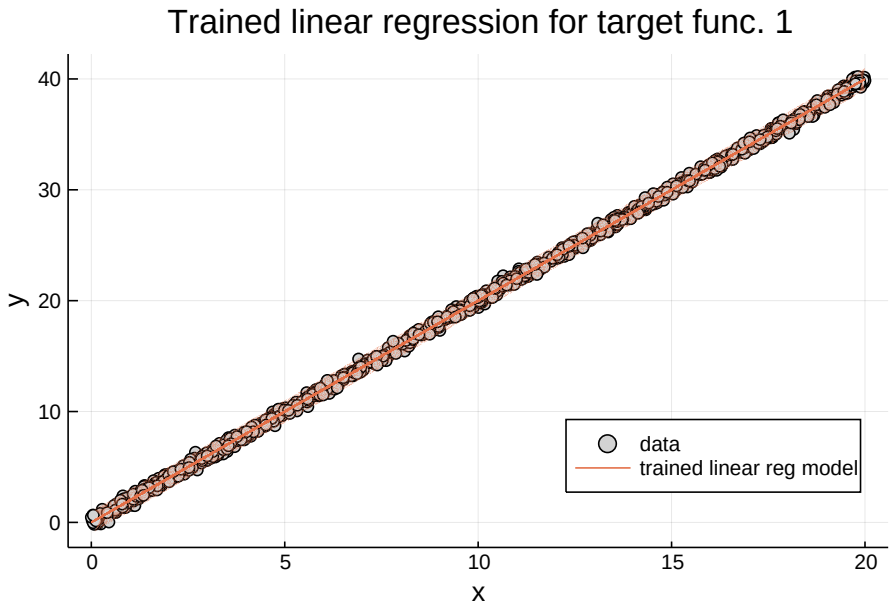
   and, after all iterations, returns the final estimate of β.

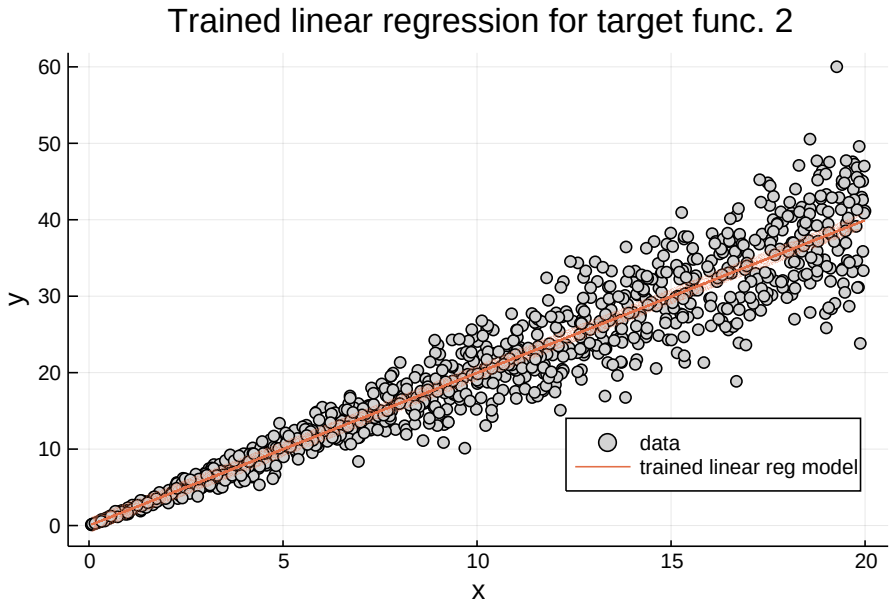
```
using Logging # Print training progress to REPL, not pdf

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)
      @info "loss: $(lr_model_nll(β_curr,x,y;σ = σ_model))  β: $β_curr"

      # Computing gradient
      grad_β = gradient(β -> lr_model_nll(β,x,y;σ=σ_model),β_curr)[1]

      # Updating estimate with gradient descent
      β_curr = β_curr - (grad_β * lr)
    end
    return β_curr
end
```

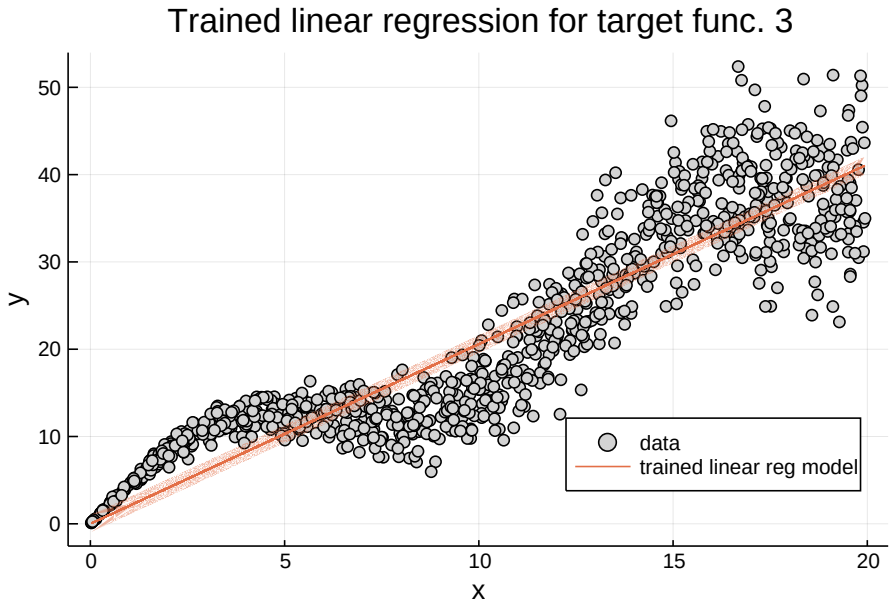
```
train_lin_reg (generic function with 1 method)
```

2. [2pts] For each target function, we 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.

```
β_init = 1000*randn() # Initial parameter

# Learned parameter
β_learned_1 = train_lin_reg(target_f1, β_init; bs= 100, lr = 1e-6, iters=1000, σ_model =
1.)

# Plotting sample of data and linear regression model with shaded region for
uncertainty
plot_1 = plot(x1[1,:], y1, seriestype=:scatter, color = [:lightgray], legend =
:bottomright,title="Trained linear regression for target func. 1", xlabel="x",
ylabel="y")
plot!(plot_1, x1[1,:],(β_learned_1 .* x1)[1,:],ribbon = [-1, 1])

# Learned parameter 2
β_learned_2 = train_lin_reg(target_f2, β_init; bs= 100, lr = 1e-6, iters=1000, σ_model =
1.)

# Plotting sample 2 of data and linear regression model with shaded region for
uncertainty
plot_2 = plot(x2[1,:], y2, seriestype=:scatter, color = [:lightgray], legend =
:bottomright,title="Trained linear regression for target func. 2", xlabel="x",
ylabel="y")
plot!(plot_2, x2[1,:],(β_learned_2 .* x2)[1,:],ribbon = [-1, 1])

# Learned parameter 3
β_learned_3 = train_lin_reg(target_f3, β_init; bs= 100, lr = 1e-6, iters=1000, σ_model =
1.)

# Plotting sample 3 of data and linear regression model with shaded region for
uncertainty
plot_3 = plot(x3[1,:], y3, seriestype=:scatter, color = [:lightgray], legend =
:bottomright,title="Trained linear regression for target func. 3", xlabel="x",
ylabel="y")
plot!(plot_3, x3[1,:],(β_learned_3 .* x3)[1,:],ribbon = [-1, 1])
```


Trained linear regression for target func. 1
y
x
data
trained linear reg model


Trained linear regression for target func. 2
y
x
data
trained linear reg model


Trained linear regression for target func. 3
y
x
data
trained linear reg model

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] The code for a fully-connected neural network (multi-layer perceptron) with one 10-dimensional hidden layer and a `tanh` nonlinearirty.

   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)
  return vec(sum((θ[4] .+ (θ[3]' .* tanh.(((θ[1])' * x) .+ θ[2]))), dims=1))
end

# Random initial Parameters
θ = (rand(1,10),rand(10,),rand(1,10),rand(10,))

@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 $\sigma = 1.0$

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

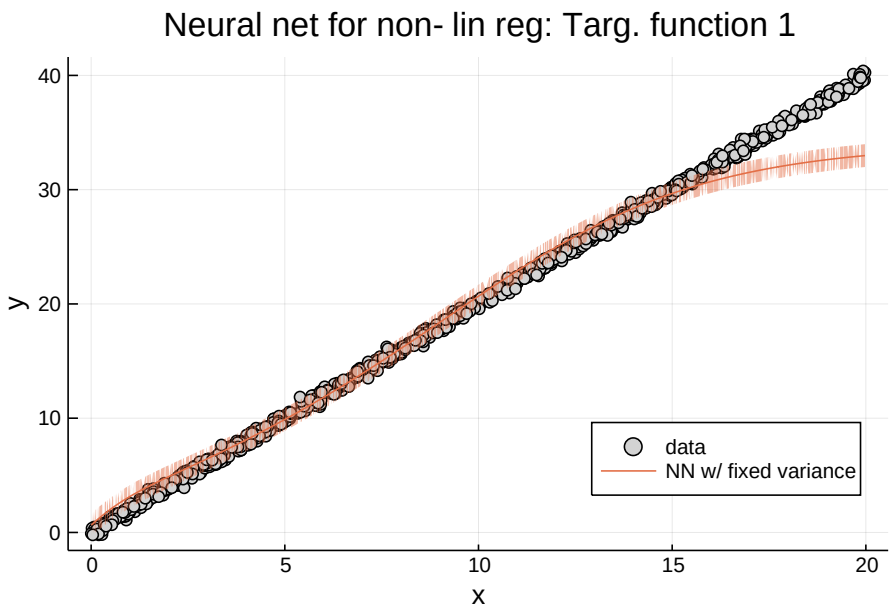
and, after all iterations, returns the final estimate of θ.

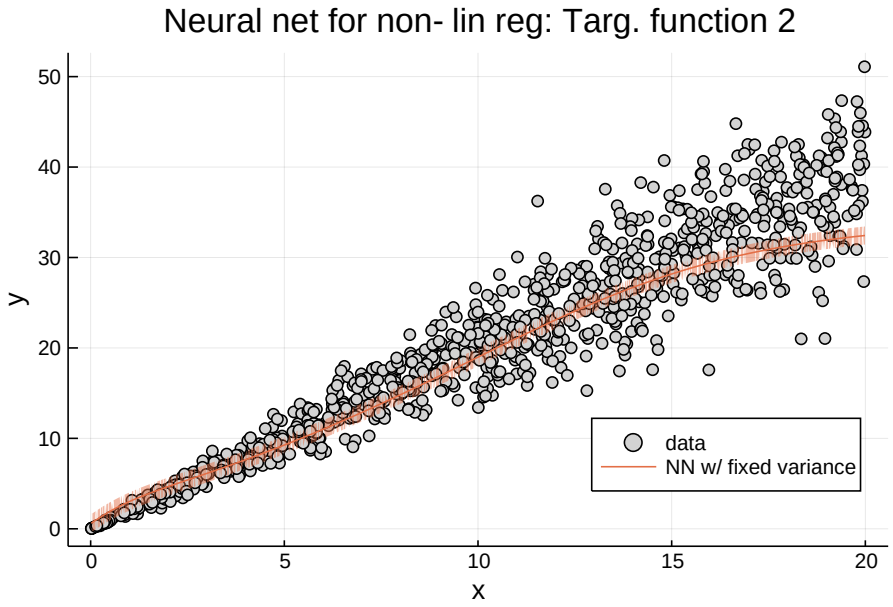
```
using Logging # Print training progress to REPL, not pdf

function train_nn_reg(target_f, θ_init; bs= 100, lr = 1e-5, iters=1000, σ_model = 1. )
    θ_curr = θ_init
    for i in 1:iters
      x,y = sample_batch(target_f,bs)
      @info "loss: $(nn_model_nll(θ_init,x,y;σ = σ_model))"

      # Computing gradient
      grad_θ = gradient(θ -> nn_model_nll(θ,x,y;σ=σ_model),θ_curr)[1]

      # Updating the estimate with gradient descent
      θ_curr = θ_curr .- (grad_θ .* lr)
    end
    return θ_curr
end
```

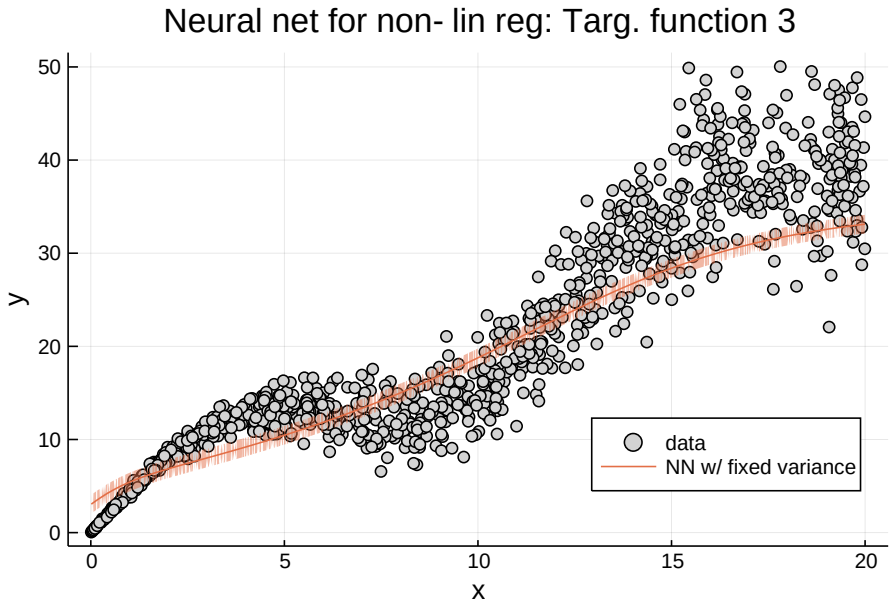
```
train_nn_reg (generic function with 1 method)
```

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 $\sigma = 1.0$

```
θ_init = (rand(1,10),rand(10,),rand(1,10),rand(10,))

x_nn1, y_nn1 = sample_batch(target_f1,1000)
θ_learned_1 = train_nn_reg(target_f1, θ_init; bs= 100, lr = 1e-5, iters=1000, σ_model = 1.)
plot_1 = plot(x_nn1[1,:], y_nn1, seriestype=:scatter, color = [:lightgray], legend =
:bottomright,title="Neural net for non-lin reg: Targ. function 1", xlabel="x",
ylabel="y")
plot!(plot_1, x_nn1[1,:],(neural_net(x_nn1,θ_learned_1)),ribbon = [-1,1],seriestype=:line)

x_nn2, y_nn2 = sample_batch(target_f2,1000)
θ_learned_2 = train_nn_reg(target_f2, θ_init; bs= 100, lr = 1e-5, iters=1000, σ_model = 1.)
plot_2 = plot(x_nn2[1,:], y_nn2, seriestype=:scatter, color = [:lightgray], legend =
:bottomright,title="Neural net for non-lin reg: Targ. function 2", xlabel="x",
ylabel="y")
plot!(plot_2, x_nn2[1,:],(neural_net(x_nn2,θ_learned_2)),ribbon = [-1,1],seriestype=:line)

x_nn3, y_nn3 = sample_batch(target_f3,1000)
θ_learned_3 = train_nn_reg(target_f3, θ_init; bs= 100, lr = 1e-5, iters=1000, σ_model = 1.)
plot_3 = plot(x_nn3[1,:], y_nn3, seriestype=:scatter, color = [:lightgray], legend =
:bottomright,title="Neural net for non-lin reg: Targ. function 3", xlabel="x",
ylabel="y")
plot!(plot_3, x_nn3[1,:],(neural_net(x_nn3,θ_learned_3)),ribbon = [-1,1],seriestype=:line)
```


Neural net for non- lin reg: Targ. function 1
y
x
data
NN w/ fixed variance


Neural net for non- lin reg: Targ. function 2
y
x
data
NN w/ fixed variance


Neural net for non- lin reg: Targ. function 3
y
x
data
NN w/ fixed variance

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] 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)
  return (vec(sum((θ[4] .+ (θ[3]' .* tanh.(((θ[1])' * x) .+ θ[2]))), dims=1)),
vec(sum((θ[8] .+ (θ[7]' .* tanh.(((θ[5])' * x) .+ θ[6]))), dims=1)))
end

# Random initial Parameters
θ = (rand(1,10),rand(10,),rand(1,10),rand(10,),rand(1,10),rand(10,),rand(1,10),rand(10,))

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

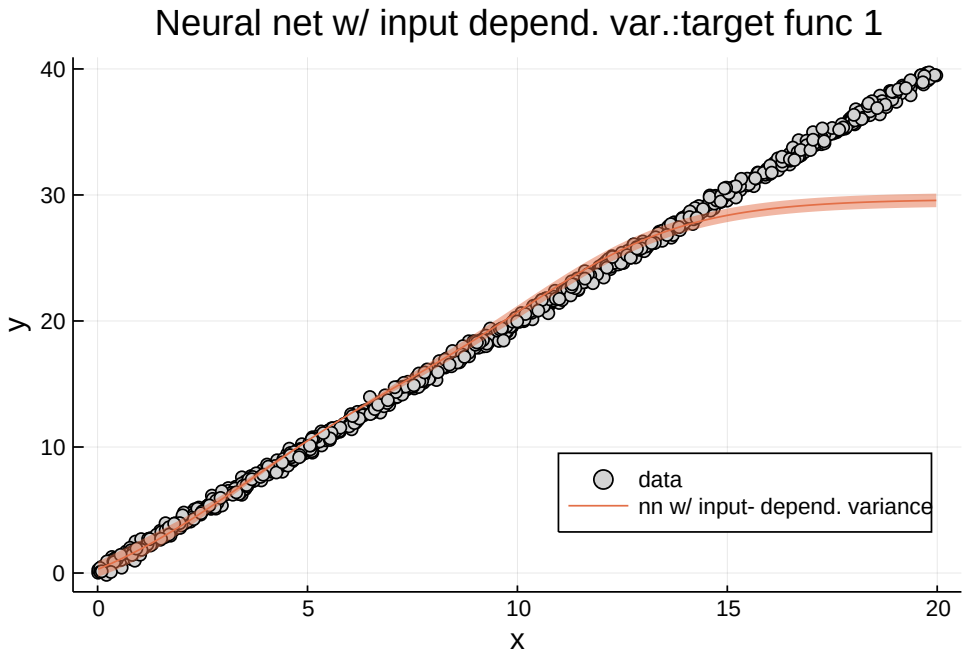
```
function train_nn_w_var_reg(target_f, θ_init; bs= 100, lr = 1e-4, iters=10000)
    θ_curr = θ_init
    for i in 1:iters
      x,y = sample_batch(target_f,bs)
      @info "loss: $(nn_with_var_model_nll(θ_init,x,y))"

      # Compute gradients
      grad_θ = gradient(θ -> nn_with_var_model_nll(θ,x,y),θ_curr)[1]

      # Update estimate with gradient descent
      θ_curr = θ_curr .- (grad_θ .* lr)
    end
    return θ_curr
end
```

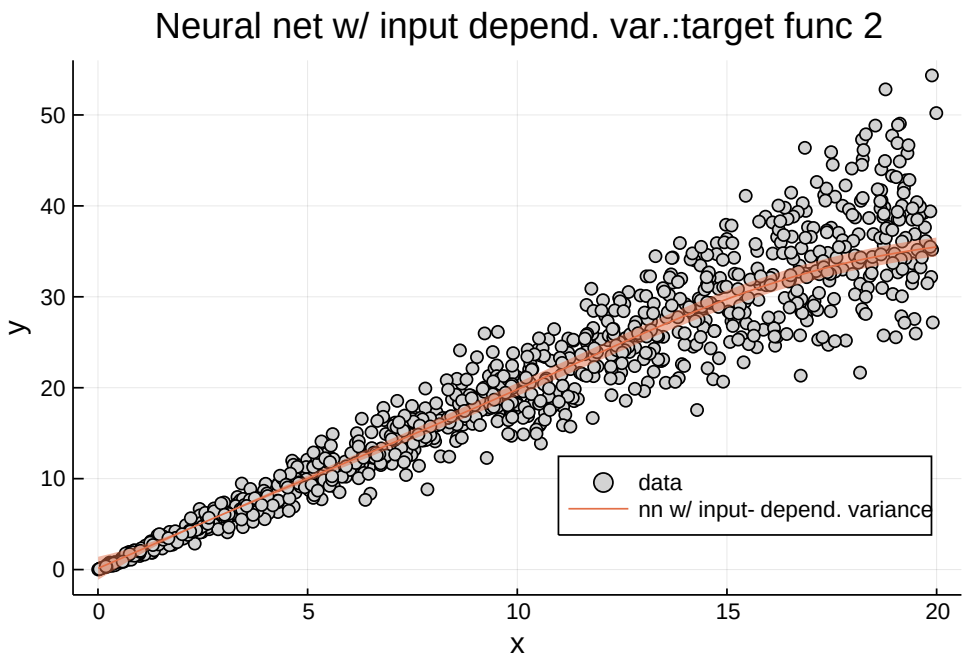
```
train_nn_w_var_reg (generic function with 1 method)
```

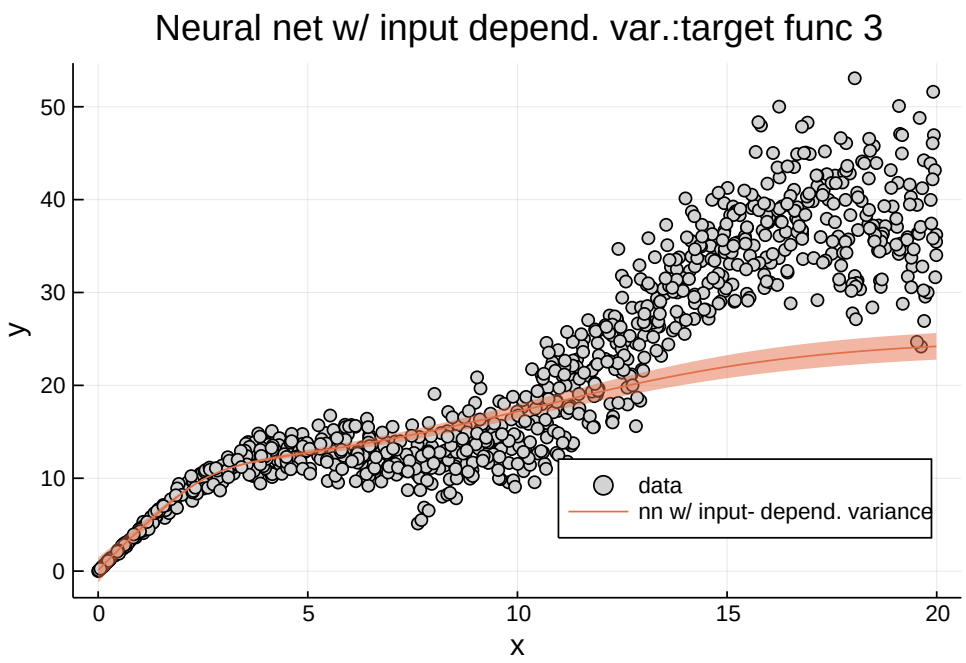
4. [4pts] For each target function, start with an initialization of the network parameters, θ, learn an estimate for θ_{learned} by gradient descent. Then plot a $n = 1000$ sample of the dataset and the learned regression model with shaded uncertainty bounds corresponding to plus/minus one standard deviation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

```
# Initial Parameters
θ_init_nn =
(rand(1,10),rand(10,),rand(1,10),rand(10,),rand(1,10),rand(10,),rand(1,10),rand(10,))

θ_learned_1_nn = train_nn_w_var_reg(target_f1, θ_init_nn; bs= 100, lr = 1e-4, iters=10000)
x_nn_11, y_nn_11 = sample_batch(target_f1,1000)
a = neural_net_w_var(x_nn_11,θ_learned_1_nn)
plot_1 = plot(x_nn_11[1,:], y_nn_11, seriestype=:scatter, color = [:lightgray], legend =
:bottomright,title="Neural net w/ input depend. var.:target func 1", xlabel="x",
ylabel="y")
plot!(plot_1, x_nn_11[1,:],(a[1]),ribbon = [-a[2],a[2]],seriestype=:line)

θ_learned_2_nn = train_nn_w_var_reg(target_f2, θ_init_nn; bs= 100, lr = 1e-4, iters=10000)
```

```
x_nn_22, y_nn_22 = sample_batch(target_f2,1000)
a2 = neural_net_w_var(x_nn_22,θ_learned_2_nn)
plot_2 = plot(x_nn_22[1,:], y_nn_22, seriestype=:scatter, color = [:lightgray], legend =
:bottomright,title="Neural net w/ input depend. var.:target func 2", xlabel="x",
ylabel="y")
plot!(plot_2, x_nn_22[1,:],(a2[1]),ribbon = [-a2[2],a2[2]],seriestype=:line)

θ_learned_3_nn = train_nn_w_var_reg(target_f3, θ_init_nn; bs= 100, lr = 1e-4, iters=10000)
x_nn_33, y_nn_33 = sample_batch(target_f3,1000)
a3 = neural_net_w_var(x_nn_33,θ_learned_3_nn)
plot_3 = plot(x_nn_33[1,:], y_nn_33, seriestype=:scatter, color = [:lightgray], legend =
:bottomright,title="Neural net w/ input depend. var.:target func 3", xlabel="x",
ylabel="y")
plot!(plot_3, x_nn_33[1,:],(a3[1]),ribbon = [-a3[2],a3[2]],seriestype=:line)
```






Neural net w/ input depend. var.:target func 3
y
x
0
10
20
30
40
50
0
5
10
15
20
data
nn w/ input- depend. variance