Table of Contents

CSC 2506 Probabilistic Learning and Reasoning

Assignment I - Q3

Tianyi Liu liuti110 1005820827

Regression [30pts]

Manually Derived Linear Regression

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. What happens if $n < m$?
2. What are the expectation and covariance matrix of $\hat{\beta}$, for a given true value of β?
3. Show that maximizing the likelihood is equivalent to minimizing the squared error $\sum_{i=1}^{n}(y_i - x_i\beta)^2$. [Hint: Use $\sum_{i=1}^{n} a_i^2 = a^T a$]
4. 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]
5. 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:

1. When $n < m$, this is an under-determined case in which there are more features than number of equations, i.e., there are infinitely many solutions.
2. We write $Y = X^T\beta + n$, where n is the noise vector, which conforms the distribution of Y.

$$
\begin{aligned}
\mathbb{E}[\hat{\beta}] &= \mathbb{E}[(XX^T)^{-1}XY] \\
&= \mathbb{E}[(XX^T)^{-1}X(X^T\beta + n)] \\
&= \mathbb{E}[(XX^T)^{-1}XX^T\beta + (XX^T)^{-1}Xn] \\
&= \mathbb{E}[\beta] + \mathbb{E}[(XX^T)^{-1}Xn] \\
&= \beta + (XX^T)^{-1}\mathbb{E}[Xn] \\
&= \beta + (XX^T)^{-1}\mathbb{E}[X]\mathbb{E}[n] \\
&= \beta.
\end{aligned}
$$

$$
\begin{aligned}
Cov[\hat{\beta}] &= \mathbb{E}[(\hat{\beta} - \mathbb{E}[\beta])(\hat{\beta} - \mathbb{E}[\beta])^T] \\
&= \mathbb{E}[((XX^T)^{-1}Xn)((XX^T)^{-1}Xn)^T] \\
&= \mathbb{E}[(XX^T)^{-1}Xnn^TX^T(XX^T)^{-T}] \\
&= (XX^T)^{-1}X\mathbb{E}[nn^T]X^T(XX^T)^{-T} \\
&= (XX^T)^{-1}X\sigma^2IX^T(XX^T)^{-T} \\
&= \sigma^2(XX^T)^{-1}XX^T(XX^T)^{-T} \\
&= \sigma^2(XX^T)^{-T} \\
&= \sigma^2(XX^T)^{-1}.
\end{aligned}
$$

3.

$$
\begin{aligned}
l(Y \mid X, \beta, \sigma) &= \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(Y - X^T\beta)^T\Sigma^{-1}(Y - X^T\beta)\right) \\
&= \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2\sigma^2}(Y - X^T\beta)^T(Y - X^T\beta)\right) \\
&= \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - x_i\beta)^2\right)
\end{aligned}
$$

Hence, maximizing the likelihood is equivalent to minimizing the exponent of likelihood, which is in turns, minimizing the squared error. □

4.

$$
\begin{aligned}
\mathcal{L}_{MSE} &= \sum_{i=1}^{n}(y_i - x_i\beta)^2 \\
&= (Y - X^T\beta)^T(Y - X^T\beta) \\
&= Y^TY + \beta^TXX^T\beta - 2Y^TX^T\beta
\end{aligned}
$$

5.

We start from the expression in 3.

$$-\log l(D \mid X, \beta, \sigma) = -\log\left(\frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}}\exp\left(-\frac{1}{2\sigma^2}(Y - X^T\beta)^T(Y - X^T\beta)\right)\right)$$

$$\begin{aligned}\frac{\partial -\log l(D \mid X, \beta, \sigma)}{\partial \beta} &= \frac{\partial \frac{1}{2\sigma^2}(Y - X^T\beta)^T(Y - X^T\beta)}{\partial \beta} \\ &= \frac{1}{2\sigma^2}(2XX^T\beta - 2XY).\end{aligned}$$

Set $\frac{\partial -\log l(D|X,\beta,\sigma)}{\partial \beta}$ to 0 yields:

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

Toy Data

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
```

```
target_f1 (generic function with 2 methods)
```

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

```
target_f2 (generic function with 1 method)
```

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

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

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

Sample data from the target functions

Write a function which produces a batch of data $x \sim \mathrm{Uniform}(0, 20)$ and `y = target_f(x)`

```
using Distributions
```

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

```
function sample_batch(target_f, batch_size)
  x1 = reshape(rand(Uniform(0,20), batch_size), (1, batch_size))
  x = []
  for i = 1:batch_size
    append!(x, 20 * rand())
  end
  x = reshape(x, (1, batch_size))
  x = convert.(Float64, x)
  y = target_f(x)
  return (x,y)
end
```

Test assumptions about your dimensions

```
using Test
```

```
Test.DefaultTestSet("sample dimensions are correct", Any[], 6, false)
```

```
begin
    @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
end
```

Plot the target functions

For all three targets, plot a $n = 1000$ sample of the data.

Note: You will use these plots later, in your writeup. Conmsider suppressing display once other questions are complete.

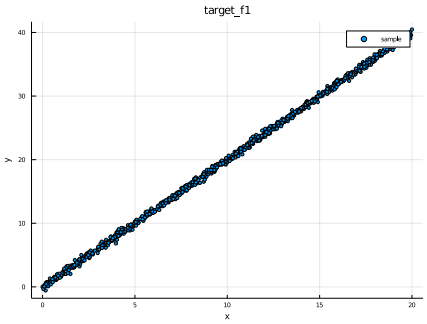
```
using Plots
```

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

```
# generic plot function
function target_plot(x, y, title)
    p = scatter(dropdims(x, dims=1), y, label="sample", size=(800,600));
    #scatter!(1:1000, y, label="y");
    xlabel!("x");
    ylabel!("y");
    title!(title);
    return p
end
```

```
(1×1000 Array{Float64,2}:
 19.9986  18.5136  17.2244  3.29698  12.9442  …  18.1044  19.0097  8.72553  12.0572,
```

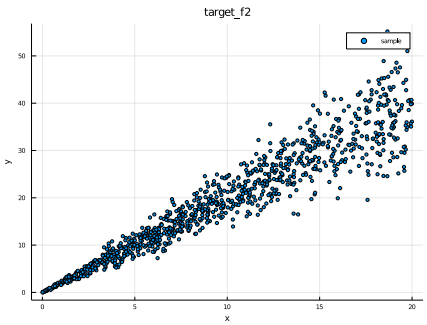
```
# x1,y1  #TODO
begin
    (x1, y1) = sample_batch(target_f1, 1000);
end
```




```
# plot_f1  #TODO
begin
    graph_f1 = target_plot(x1,y1,"target_f1");
end
```

```
(1×1000 Array{Float64,2}:
 17.5299  8.95198  16.9234  17.318  8.68084  …  17.5431  9.28441  1.01516  0.387442,
```

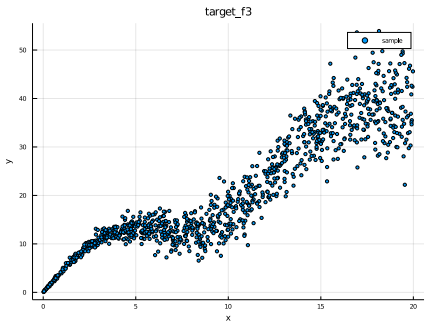
```
# x2,y2 #TODO
begin
    (x2, y2) = sample_batch(target_f2, 1000);
end
```



```
# plot_f2 #TODO
begin
    graph_f2 = target_plot(x2,y2,"target_f2");
end
```

```
(1×1000 Array{Float64,2}:
  7.74373  10.2027  14.7628  11.0315  17.2474  …  19.4715  7.76624  19.1387  8.55929)
```

```
# x3,y3  #TODO
begin
    (x3, y3) = sample_batch(target_f3, 1000);
end
```




```
# plot_f3  #TODO
begin
    graph_f3 = target_plot(x3,y3,"target_f3");
end
```

Linear Regression Model with $\hat{\beta}$ MLE

Code the hand-derived MLE

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.

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

```
function beta_mle(X,Y)
  beta = inv(X * X') * X * Y
  return beta[1]
end
```

```
# n=1000 # batch_size
# This code cell has been commented and previously generated data are used here
directly
```

```
# x_1, y_1  #TODO
# Use above-defined data directly
```

```
1.9996558893098424
```

```
# β_mle_1  #TODO
begin
    β_mle_1 = beta_mle(x1,y1);
end
```

```
# x_2, y_2  #TODO
# Use above-defined data directly
```

```
1.9833990377537423
```

```
# β_mle_2  #TODO
begin
    β_mle_2 = beta_mle(x2,y2);
end
```

```
# x_3, y_3 #TODO
# Use above-defined data directly
```

```
2.0675136708392907
```

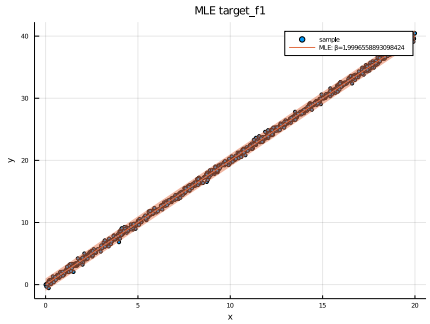
```
# β_mle_3  #TODO
begin
    β_mle_3 = beta_mle(x3,y3);
end
```

Plot the MLE linear regression model

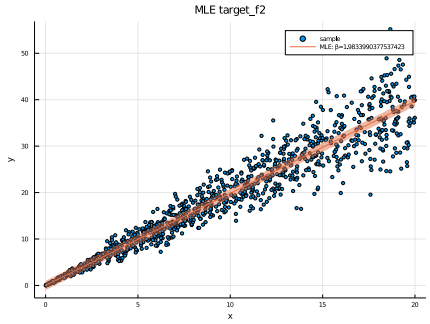
For each function, plot the linear regression model given by $Y \sim \mathcal{N}(X^T\hat{\beta}, \sigma^2 I)$ for $\sigma = 1.$. This plot should have the line of best fit given by the maximum likelihood estimate, as well as a shaded region around the line corresponding to plus/minus one standard deviation (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**

```
target_plot_2 (generic function with 1 method)
```

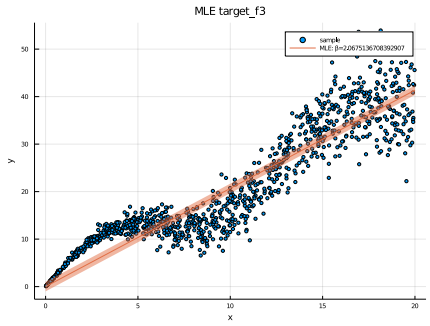
```
function target_plot_2(p, β, title, label)
    plot(p);
    plot!(0:0.5:20, collect(0:0.5:20) * β, ribbon=1.0, label=label, size=
(800,600));
    title!(title);
    return current();
end
```




```
# plot!(plot_f1,TODO)
begin
    target_plot_2(target_plot(x1,y1,"target_f1"), β_mle_1[1], "MLE target_f1",
"MLE: β=$β_mle_1");
end
```




```
# plot!(plot_f2,TODO)
begin
    target_plot_2(target_plot(x2,y2,"target_f2"), β_mle_2[1], "MLE target_f2",
"MLE: β=$β_mle_2");
end
```




```
# plot!(plot_f3,TODO)
begin
    target_plot_2(target_plot(x3,y3,"target_f3"), β_mle_3[1], "MLE target_f3",
"MLE: β=$β_mle_3");
end
```

Log-likelihood of Data Under Model

Code for Gaussian Log-Likelihood

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

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

```
begin
    function gaussian_log_likelihood(μ, σ, x)
      """
      compute log-likelihood of x under N(μ,σ)
      """
        function scalar_gaussian_lllhd(μs, σs, xs)
            # Numerical unstable implementation
            # return log(1 / √(2 * π * σs ^ 2) * e ^ (-0.5 * ((xs - μs) / σs) ^ 2))
            return log(1 / √(2 * π * σs ^ 2)) + (-0.5 * ((xs - μs) / σs) ^ 2)
        end
        return scalar_gaussian_lllhd.(μ, σ, x)
    end
end
```

Test Gaussian likelihood against standard implementation

```
Test.DefaultTestSet("Gaussian log likelihood", Any[], 6, false)
```

```
begin
    @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) ≈ logpdf.(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) ≈ logpdf.(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) ≈ logpdf.(Normal.(μ,σ),x) # Correct
Values
    @show gaussian_log_likelihood.(μ,σ,x)
        @show logpdf.(Normal.(μ,σ),x)
    end
end
```

Model Negative Log-Likelihood

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.

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

```
function lr_model_nll(β,x,y;σ=1.)
    #TODO: Negative Log Likelihood
    return - sum(gaussian_log_likelihood(x' * β, σ, y))
end
```

Compute Negative-Log-Likelihood on data

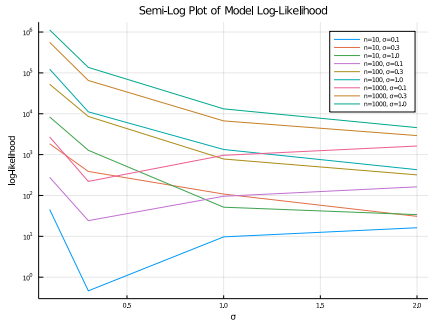
Use this function to compute and report the negative-log-likelihood of a $n \in \{10, 100, 1000\}$ batch of data under the model with the maximum-likelihood estimate $\hat{\beta}$ and $\sigma \in \{0.1, 0.3, 1., 2.\}$ for each target function.

```
begin
    """
    # Re-implement numerically stable solution
    # function gaussian_log_likelihood_stable(μ, σ, x)
    """
    # compute log-likelihood of x under N(μ,σ)
    """
        function scalar_gaussian_lllhd_stable(μs, σs, xs)
            return log(1 / √(2 * π * σs ^ 2)) + (-0.5 * ((xs - μs) / σs) ^ 2)
        end
        return scalar_gaussian_lllhd_stable.(μ, σ, x)
    end

    function lr_model_nll_stable(β,x,y;σ=1.)
        #TODO: Negative Log Likelihood
        return - sum(gaussian_log_likelihood_stable(x' * β, σ, y))
    end
    """

    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
end
```

Effect of model variance

For each target function, what is the best choice of σ?

Please note that σ and batch-size n are modelling hyperparameters. In the expression of maximum likelihood estimate, σ or n do not appear, and 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. The take-home is that you cannot directly compare the negative log-likelihoods achieved by these models with different hyperparameter settings.



`target_f1`

- Best overall configuration: $n = 10, \sigma = 0.3$;
- Best configuration when $n = 10$: $\sigma = 0.3$;
- Best configuration when $n = 100$: $\sigma = 0.3$;
- Best configuration when $n = 1000$: $\sigma = 0.3$.

`target_f2`

- Best overall configuration: $n = 10, \sigma = 2.0$;
- Best configuration when $n = 10$: $\sigma = 2.0$;
- Best configuration when $n = 100$: $\sigma = 2.0$;
- Best configuration when $n = 1000$: $\sigma = 2.0$.

`target_f3`

- Best overall configuration: $n = 10, \sigma = 2.0$;
- Best configuration when $n = 10$: $\sigma = 2.0$;
- Best configuration when $n = 100$: $\sigma = 2.0$;
- Best configuration when $n = 1000$: $\sigma = 2.0$.

Automatic Differentiation and Maximizing Likelihood

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.

Compute Gradients with AD, Test against hand-derived

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
```

```
Test.DefaultTestSet("Gradients wrt parameter", Any[], 1, false)
```

```
begin
    @testset "Gradients wrt parameter" begin
    β_test = randn()
    σ_test = rand()
    x_ad, y_ad = sample_batch(target_f1,100)
    ad_grad = gradient((dβ, dx, dy, dσ) -> lr_model_nll(dβ, dx, dy, σ=dσ), β_test, x_ad, y_ad, σ_test);
    hand_derivative =  ((x_ad * x_ad' * β_test .- x_ad * y_ad) ./ (σ_test ^ 2))[1];
    @test ad_grad[1] ≈ hand_derivative
    end
end
```

Train Linear Regression Model with Gradient Descent

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.

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.

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

```
begin
    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)
            #TODO: log loss, if you want to monitor training progress
            lllhd = lr_model_nll(β_curr, x, y, σ=σ_model)
            @info "iter: $i/$iters\tloss: $lllhd\tβ: $β_curr"
            #TODO: compute gradients
            grad_β = gradient(dβ->lr_model_nll(dβ, x, y, σ=σ_model), β_curr)
            #TODO: gradient descent
            β_curr -= lr * grad_β[1]
        end
        return β_curr
    end
end
```

Parameter estimate by gradient descent

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.

```
β_init = 196.78156589784368
```

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

```
Float64[2.00012, 1.99946, 2.05634]
```

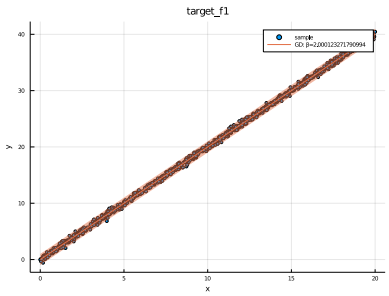
```
begin
    #TODO: call training function
    # β_learned = train_lin_reg.([target_f1, target_f2, target_f3], β_init)
    β_1_gd = train_lin_reg(target_f1, β_init, σ_model=0.3)
    β_2_gd = train_lin_reg(target_f2, β_init, iters=10000, σ_model=2.0)
    β_3_gd = train_lin_reg(target_f3, β_init, iters=10000, σ_model=2.0)
    β_learned = [β_1_gd, β_2_gd, β_3_gd]
end
```

Note that we have already seen that the best σ's for `target_f1`, `target_f2`, `target_f3` are $0.3, 2.0, 2.0$ correspondingly. Therefore, we use the best σ here.

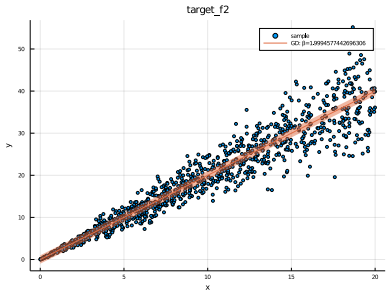
Plot learned models

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.

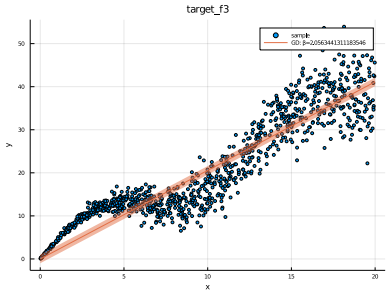
```
Plots.Plot{Plots.GRBackend}[
1:
```




```
2:
```




```
3:
```




```
]
```

```
#TODO: For each target function, plot data samples and learned regression
begin
    β_1, β_2, β_3 = β_learned[1], β_learned[2], β_learned[3]
    graph_base_gd = [target_plot(x1,y1,""), target_plot(x2,y2,""),
target_plot(x3,y3,"")]
    title_base = ["target_f1", "target_f2", "target_f3"]
    label_base_gd = ["GD: β=$β_1", "GD: β=$β_2", "GD: β=$β_3"]
    target_plot_2.(graph_base_gd, β_learned, title_base, label_base_gd)
end
```

Non-linear Regression with a Neural Network

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).

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

Fully-connected Neural Network

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_net (generic function with 1 method)
```

```
# Neural Network Function
function neural_net(x,θ)
    z1 = tanh.(x' * θ[1] .+ θ[2])
    z2 = z1 * θ[3] .+ θ[4]
    return z2 #TODO
end
```

```
begin
    # Random initial Parameters
    h = 10;
    θ = [randn(1, h), randn(1, h), randn(h), randn(1)]; #TODO
end;
```

Test assumptions about model output

Test, at least, the dimension assumptions.

```
Test.DefaultTestSet("neural net mean vector output", Any[], 1, false)
```

```
begin
    @testset "neural net mean vector output" begin
    n_nn = 100
    x_nn,y_nn = sample_batch(target_f1,n_nn)
    μ_nn = neural_net(x_nn,θ)
    @test size(μ_nn) == (n_nn,)
    end
end
```

Negative Log-likelihood of NN model

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 $\sigma = 1.0$

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

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

Training model to maximize likelihood

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.

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

```
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)
        lllhd = nn_model_nll(θ_curr, x, y, σ=σ_model)
        #TODO: log loss, if you want to montior training
        if i % 500 == 0
            @info "iter: $i/$iters\tloss: $lllhd"
        end
        #TODO: compute gradients
        grad_θ = gradient((dθ, dx, dy, dσ) -> nn_model_nll(dθ, dx, dy, σ=dσ),
θ_curr, x, y, σ_model)
        #TODO: gradient descent
        θ_curr -= lr * grad_θ[1]
    end
    return θ_curr
end
```

Learn model parameters

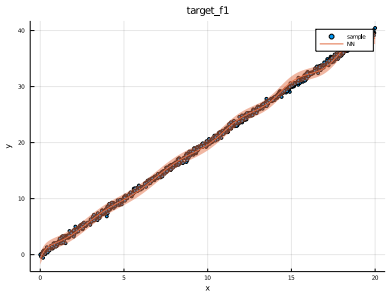
For each target function, start with an initialization of the network parameters, θ, use your train function to minimize the negative log-likelihood and find an estimate for θ_{learned} by gradient descent.

```
begin
    θ_init = [
        [rand(1, h), rand(1, h), rand(h), rand(1)],
        [rand(1, h), rand(1, h), rand(h), rand(1)],
        [rand(1, h), rand(1, h), rand(h), rand(1)]];
    θ_1_nn = train_nn_reg(target_f1, θ_init[1], iters=30000, σ_model=0.3);
    θ_2_nn = train_nn_reg(target_f2, θ_init[2], iters=30000, σ_model=2.0);
    θ_3_nn = train_nn_reg(target_f3, θ_init[3], iters=30000, σ_model=2.0);
    θ_learned_nn = [θ_1_nn, θ_2_nn, θ_3_nn]
end;
```

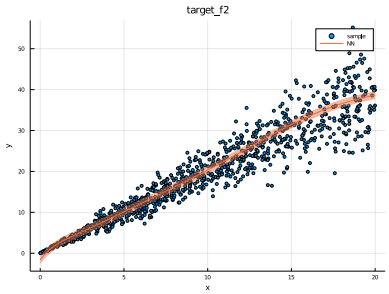
Plot neural network regression

Then plot a $n = 1000$ sample of the data and the learned regression model with shaded uncertainty bounds given by $\sigma = 1.0$

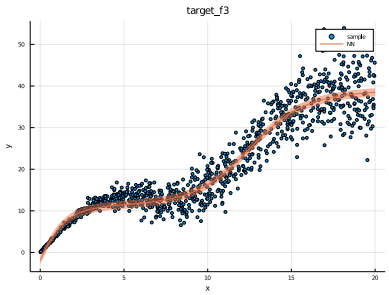
```
Plots.Plot{Plots.GRBackend}[
1:
```




```
2:
```




```
3:
```




```
]
```

```
begin
    function target_plot_3(p, θ, title, label)
        plot(p);
        plot!(0:0.1:20, neural_net(reshape(collect(0:0.1:20),
(1,size(collect(0:0.1:20))[1])), θ), ribbon=1.0, label=label);
        title!(title);
        return current();
    end

    graph_base_nn = [target_plot(x1,y1,""), target_plot(x2,y2,""),
target_plot(x3,y3,"")]
    label_base_nn = ["NN", "NN", "NN"]
    target_plot_3.(graph_base_nn, θ_learned_nn, title_base, label_base_nn)
end
```

Input-dependent Variance

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.

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

Neural Network that outputs log-variance

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 $\log\sigma$.

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

```
# Neural Network with variance
function neural_net_w_var(x,θ)
    z1 = tanh.(x' * θ[1] .+ θ[2])
    μ = z1 * θ[3] .+ θ[4]
    logσ = z1 * θ[5] .+ θ[6]
    return μ, logσ #TODO
end
```

```
Array{Float64,N} where N[1×10 Array{Float64,2}:
                         -0.209436  -0.636089  -0.410555  …  -0.449728  -1.34406  0.4
```

```
# Random initial Parameters
begin
    #TODO
    h_μσ = 10
    θ_μσ = [randn(1, h_μσ), randn(1, h_μσ), randn(h_μσ), randn(1), randn(h_μσ),
randn(1)]; #TODO
end
```

Test model assumptions

Test the output shape is as expected.

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

```
begin
    @testset "neural net mean and logsigma vector output" begin
    n_μσ = 100
    x_μσ, y_μσ = sample_batch(target_f1, n_μσ)
    μ_μσ, logσ = neural_net_w_var(x_μσ,θ_μσ)
    @test size(μ_μσ) == (n_μσ,)
    @test size(logσ) == (n_μσ,)
    end
end
```

Negative log-likelihood with modelled variance

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 $\exp \log \sigma$)

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

```
function nn_with_var_model_nll(θ,x,y)
    μ, logσ = neural_net_w_var(x, θ)
    return - sum(gaussian_log_likelihood(μ, exp.(logσ), y))
end
```

Write training loop

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.

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

```
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) #TODO
        lllhd = nn_with_var_model_nll(θ_curr, x, y)
        #TODO: log loss
        if i % 500 == 0
            @info "iter: $i/$iters\tloss: $lllhd"
        end
        #TODO compute gradients
        grad_θ = gradient((dθ, dx, dy) -> nn_with_var_model_nll(dθ, dx, dy),
θ_curr, x, y)
        #TODO gradient descent
        θ_curr -= lr * grad_θ[1]

    end
    return θ_curr
end
```

Learn model with input-dependent variance

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.

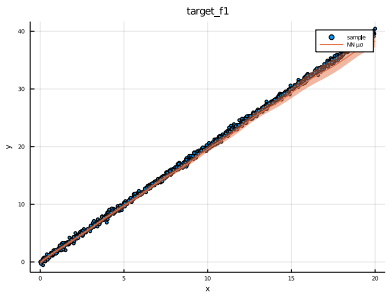
```
θ_init_μσ =
  Array{Array{Float64,N} where N,1}[Array{Float64,N} where N[1×10 Array{Float64,2}:
                                                             -7.04776e-5  4.75053e-5  3
```

```
#TODO: For each target function
θ_init_μσ = 0.0001 * [
    [randn(1, h_μσ), randn(1, h_μσ), randn(h_μσ), randn(1), randn(h_μσ), randn(1)],
    [randn(1, h_μσ), randn(1, h_μσ), randn(h_μσ), randn(1), randn(h_μσ), randn(1)],
    [randn(1, h_μσ), randn(1, h_μσ), randn(h_μσ), randn(1), randn(h_μσ), randn(1)]]
#TODO
```

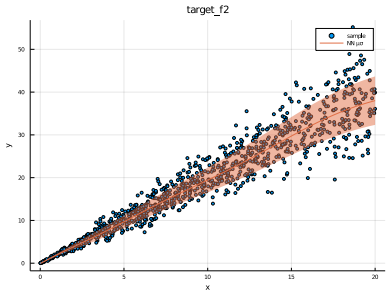
```
θ_learned_μσ = train_nn_w_var_reg.([target_f1, target_f2, target_f3], θ_init_μσ,
bs=500, lr=1e-5, iters=80000); #TODO
```

Plot model

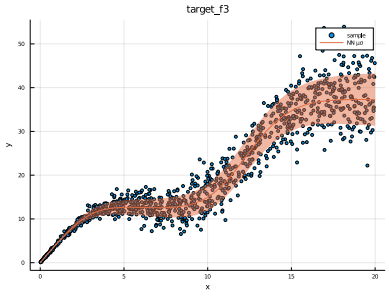
```
Plots.Plot{Plots.GRBackend}[
1:
```




```
2:
```




```
3:
```




```
]
```

```
#TODO: plot data samples and learned regression
begin
    function target_plot_4(p, θ, title, label)
        plot(p);
        μ_learned, σ_learned = neural_net_w_var(reshape(collect(0:0.1:20),
(1,size(collect(0:0.1:20))[1])), θ)
        plot!(0:0.1:20, μ_learned, ribbon=exp.(σ_learned), label=label);
        title!(title);
    return current();
end
    graph_base_μσ = [target_plot(x1,y1,""), target_plot(x2,y2,""),
target_plot(x3,y3,"")]
    label_base_μσ = ["NN μσ", "NN μσ", "NN μσ"]
    target_plot_4.(graph_base_μσ, θ_learned_μσ, title_base, label_base_μσ)
end
```

Answer: The results here is reasonable. For `target_f1`, the underlying varaince is uniform, so as the learned one. For `target_f2`, `target_f3`, the underlying variances are increasing with x, so as the learned variances.

Spend time making it better (optional)

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.