Linear Regression in Julia by Travis McVoy

In this notebook, we will examine what I believe is the most basic form of gradient descent: linear regression.

A NOTE ABOUT CITATIONS

I imagine there's a way to use BibTex and create nice citations within the notebook. Unfortunately, I don't know how to do that, and I don't really have the time to figure it out. As an alternative, I'll do citations by author name, and then include the references at the end of the notebook.

The Theory of Least Squares

When working with least squares, our data comes as a collection of observation pairs **[Watt]** that come in the form

$$(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)$$

where each y_i is the corresponding label (yes vs. no, 0 vs. 1 etc.) for the input vector

$$\mathbf{x}_i = \begin{bmatrix} x_{1,i} \\ x_{2,i} \\ x_{3,i} \\ \vdots \\ x_{m,i} \end{bmatrix}.$$

We wish to separate this data, which we can do by fitting a hyperplane in an $m+1$ dimensional space (m dimensions for input, 1 dimension for bias). In the simplest case ($m = 1$) the problem reduces to fitting a line, which is what we'll do here.

In order to fit the hyper plane, we wish to satisfy the equation

$$w_0 + \mathbf{x}_i^T \mathbf{w} \approx y_i$$

which can be written more compactly by mapping $\mathbf{x}_i$ to

$$\mathring{\mathbf{x}} = \begin{bmatrix} 1 \\ x_{1,i} \\ x_{2,i} \\ x_{3,i} \\ \vdots \\ x_{m,i} \end{bmatrix}$$

and $\mathbf{w}$ to

$$\mathring{\mathbf{w}} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_m \end{bmatrix}.$$

It then follows that $w_0 + \mathbf{x}_i^T \mathbf{w} \approx y_i$ is equivalent to

$$\mathring{\mathbf{x}}_i^T \mathring{\mathbf{w}} \approx y_i.$$

In pratice, some people will map so that the bias is stored in the weight vector, and others will not. It's usually clear from context, but I will make extra efforts to indicate which I'm doing. At this point, we have an equation, but there's nothing that promises that our initial weight vector implies the equation above holds for all input instances (and in fact, that will probably never be the case). To fix this, we use what's called *gradient descent*.

Gradient Descent

In single variable calculus, we saw via Rolle's Theorem that we can minimize (or maximize) a function by setting its derivative equal to zero. Naturally, there is a multivariate extension; for a vector valued function f, the gradient of f, written as ∇f, denotes the direction of steepest ascent **[Simmons]**. We can use this to fit our hyperplane. Namely, we will define a function that measures the goodness of fit of our hyperplane. In particular, notice that the smaller the magnitude of the value

$$y_i - \mathring{\mathbf{x}}_i^T \mathring{\mathbf{w}}$$

is, the better our approximation is. The astute will notice that $y_i - \mathring{\mathbf{x}}_i^T \mathring{\mathbf{w}}$ will not always be postive. That is, there are instances where our approximation produces a value larger than y_i, which results in a negative value. Since we want to minimize the magnitude of the difference between our approximation and the real value, we can square the difference to ensure that it's value is positve. Then, if we sum up all these squares, we have a measure of error, or if you prefer, a measure of loss of accuracy. However, we aren't quite done. We want this loss to be minimized for all n instances, so we will minimize the average. Writing this as a function gives

$$L_s(\mathring{\mathbf{w}}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mathring{\mathbf{x}}_i^T \mathring{\mathbf{w}})^2$$

which is called a *loss function*. Note, this function may have other names like cost, or error in other fields/texts. We will just call it a loss function.

Recall that our original goal is to satisfy $\mathring{\mathbf{x}}_i^T \mathring{\mathbf{w}} \approx y_i$ for all inputs. We do this by minimizing our loss. Since the loss is a vector valued function, we can take its gradient. Since the gradient points in the direction of steepest ascent, by traveling in the opposite direction of the gradient, we move towards a minimum of our loss function. Notice I say *a* minimum. Depending on the surface of our loss function, we might find a local minimum, not a global minimum. There is definitely more to be said about this, but we will leave that for another notebook.

Implementing Least Squares

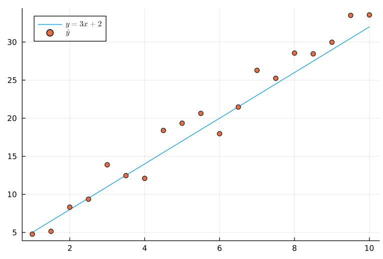
We start by generating some noisy data. In this case, we're simply going to train a model to fit to data that loosely resembles the function $y = 3x + 2$.

In [75]:

```julia
using LaTeXStrings

# <noise> let's us increase or decrease the
# noise in our data
function plotData(step, noise)
    x = hcat(1:step:10)
    y = 3 .* x .+ 2 * ones(length(x)) .+ noise * randn(length(x))
    y_line = 3 .* x .+ 2 * ones(length(x))
    plot(x,y_line, label=L"y = 3x + 2")
    display(plot!(x, y, seriestype="scatter", label=L"\hat{y}"))
    return x, y, y_line
end

x, y, y_line = plotData(0.5, 2)
println("We see above the actual line vs. the noisy data.")
println("Let's see how close we can get to the actual line using basic linear regression.")
```



```
We see above the actual line vs. the noisy data.
Let's see how close we can get to the actual line using basic linear regression.
```

Prediction model

As we saw before, our loss function has a prediction model, $\mathring{\mathbf{x}}_i^T \mathring{\mathbf{w}}$, which we will denote by $\hat{y}$. It will be nice to save our model as a function so we can check our work as we go. Note, though it is more compact symbolically to use $\mathring{\mathbf{x}}_i^T \mathring{\mathbf{w}}$, in this case it will be easier to use a bias separately due to how I've generated the data.

In [76]:

```julia
wt   = randn(1,1)   # initialize a random weight.
bias = randn(1)     # initialize a random bias.

println("Weight: ")
display(wt)
println("Bias: ")
display(bias)

model(wt, bias, x) = x * wt .+ bias

println("Compare prediction to observed: ")
display(model(wt, bias, x)[1])
display(y[1])
```

```
Weight:
1×1 Matrix{Float64}:
 -0.037010552054324485
Bias:
1-element Vector{Float64}:
 -1.0921966539877601
Compare prediction to observed:
-1.1292072060420846
4.768760414753298
```

Loss Function

Clearly, our model isn't very good right now, which is what we'd expect. Let's make it better using gradient descent. Our first step is to form a loss function.

In [77]:

```julia
function loss(wt, bias, x, y)
    ŷ = model(wt, bias, x)
    return sum((y .- ŷ).^2) / length(x)
end

# see how loss is initially:
loss(wt, bias, x, y)
```

```
Out[77]: 512.9430656843393
```

That loss is terrible, but we can fix it. We first need to compute the gradient of our loss function, which we can do using the *gradient()* function from the Flux library.

In [78]:

```julia
using Flux

dLdwt, dLdbias, _, _ = gradient(loss, wt, bias, x, y)
wt .= wt .- 0.01 .* dLdwt
bias .= bias .- 0.01 .* dLdbias
loss(wt, bias, x, y)
```

```
Out[78]: 29.872210392413056
```

Our loss decreased! And quite significantly, I might add. That sort of worries me, but let's keep training. To make iteration easier, we use a function.

In [79]:

```julia
function train(wt, bias, x, y)
    dLdwt, dLdbias, _, _ = gradient(loss, wt, bias, x, y)
    wt .= wt .- 0.01 .* dLdwt
    bias .= bias .- 0.01 .* dLdbias
end
train(wt, bias, x, y)
loss(wt, bias, x, y)
```

```
Out[79]: 4.556874836421394
```

Normally, we'd iterate from here but our loss is small enough that I think we can just call train() a few more times.

In [80]:

```julia
train(wt, bias, x, y)
loss(wt, bias, x, y)
```

```
Out[80]: 3.223797772739129
```

In [81]:

```julia
train(wt, bias, x, y)
loss(wt, bias, x, y)
```

```
Out[81]: 3.1472245486327104
```

In [82]:

```julia
train(wt, bias, x, y)
loss(wt, bias, x, y)
```

```
Out[82]: 3.1365329224790033
```

In [83]:

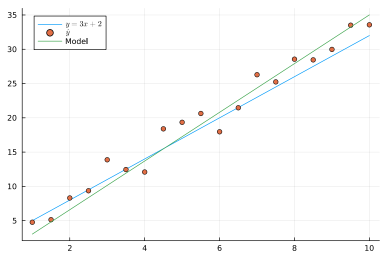
```julia
train(wt, bias, x, y)
loss(wt, bias, x, y)
```

```
Out[83]: 3.129344764990136
```

Here we see the loss is leveling out. We could try upping our step size, we could just iterate 10s of thousands of times, or we could call it could enough. Let's plot our data and see what happens.

In [84]:

```julia
plot(x,y_line, label=L"y = 3x + 2")
plot!(x, y, seriestype="scatter", label=L"\hat{y}")
display(plot!(x, model(wt, bias, x), label="Model"))
```



We can see that the line is not perfect, but it's not terrible. At this point, we could try to iterate more and get a better fit, but the change in our loss is so small that I think we would very quickly see no change at all. Alternatively, we could try again with different weight and bias values.

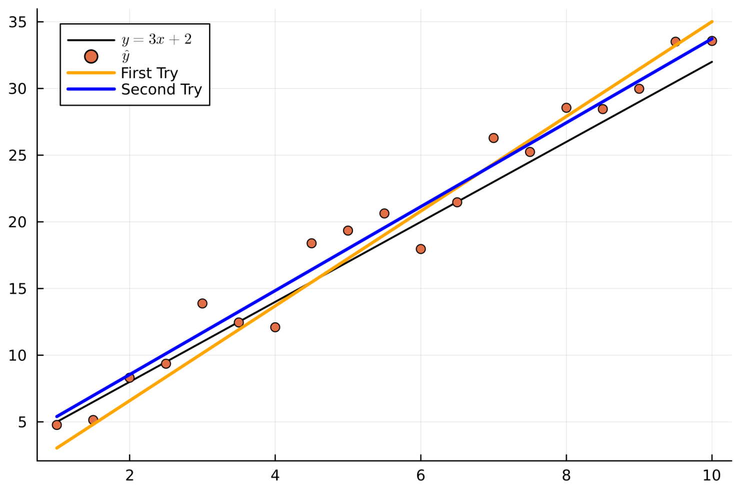
In [97]:

```julia
old_wt = wt
old_b  = bias
function tryAgain(x, y, y_line, old_wt, old_b)
    wt   = randn(1,1)   # initialize a random weight.
    bias = randn(1)     # initialize a random bias.

    for i in 1:20
        train(wt, bias, x, y)
    end

    plot(x,y_line, label=L"y = 3x + 2", lw=1.5, color="black")
    plot!(x, y, seriestype="scatter", label=L"\hat{y}")
    plot!(x, model(old_wt, old_b, x), label="First Try", color="orange", lw=2.5)
    display(plot!(x, model(wt, bias, x), label="Second Try", color="blue", lw=2.5,
            fmt = :png, dpi = 300))
    savefig("LinReg.png")
end

tryAgain(x, y, y_line, old_wt, old_b)
```



```
Out[97]: "/Users/travismcvoy/Documents/LaTeX/Website/Jupyter/LinReg.png"
```

Conclusion

It seems that we've converged to a point in which trying again doesn't change much (which makes sense given how low our loss got and how little it changed as we iterated). We now know how to implement linear regression in Julia! There's still much to do, though, as we didn't really address convexity, global vs. local minima, overshooting, or the coefficient of determination, to name a few important and related concepts. Also, we'll often we'll need better tools than linear regression when analyzing data. We'll get to better tools and the mathematics of when our gradient descent will reliably converge in future documents.

References

- *Calculus With Analytic Geometry 2nd Ed.* by George Simmons
- *Machine Learning Refined 2nd ed.* by Jeremy Watt, Reza Borhani, and Aggelos Katsaggelos
- Linear Regression documentation on Flux.jl page: https://fluxml.ai/Flux.jl/stable/tutorials/linear_regression/