Deliverables

For this assignment you need to complete the Julia programs `single_neuron_model.jl` and `single_neuron_training.jl`. For the code review you will also need to prepare (and eventually submit) a Jupyter notebook or other cleaned-up demonstration of your code. Your GitHub repository will become your submission, so commit your work as you go.

Partners and Groups

You are required to work with your assigned partner on this assignment. You are responsible for ensuring that you and your partner both fully understand all aspects of your submission; you are expected to work collaboratively and take responsibility for each other's learning.

You will also be assigned a code-review partner next week. To prepare for that code review, make sure you understand all of your code and that it is readable to people outside your group. See the code review guidelines for more information.

Objectives

This project has a couple of primary goals; it aims to help you understand:

1. single-neuron models for classification and regression, and
2. training machine learning models with gradient descent.

The one-neuron models you will be implementing are approximately equivalent to linear regression and logistic regression. Note that for both of these problems, gradient descent is not actually necessary: in a statistics class, you would derive a formula to solve for the optimal solution directly. However, gradient descent will be critically important for more complex models we will encounter soon, so we want to first learn to apply it in a simpler setting. Both linear regression and logistic regression are implemented in many existing Julia libraries; you are welcome to compare against such packages, but you may not use them as a part of your implementation.

Single-Neuron Models

In the files `single_neuron_model.jl` and `single_neuron_training.jl` there are a number of functions for you to implement (indicted by an `"unimplemented"` error).

- `linear_activation`, `linear_derivative`, `sigmoid_activation`, and `sigmoid_derivative` implement the neuron activation functions we'll be using (and their derivatives). All of these should be one-liners (or close to it).
- The two versions of `predict` each take a single-neuron model and compute the model's output(s) for a given input vector or data-set matrix.
- `MSE` computes mean-squared-error loss for an entire data set (taking predictions and targets as input).
- `accuracy` measures the fraction of points classified correctly (after rounding the outputs from a classification model).
- `gradient` takes a single-neuron model and training data, and computes the gradient of MSE w.r.t. model parameters on the training set. Note that for a neuron with W weights, this returns a
- `train!` takes a linear regression model and training data and updates the model to fit the data by repeatedly taking small steps in the $-\nabla$ direction. Each update step should be done using the `update!` function, which has been pulled out to make it easier to test that things are working correctly. The `train!` function also takes a number of optional inputs, which should be used (if a vector is provided) to give back logging information about the progress of the training.

You have been provided a constructor that generates random initial parameters for each type of linear model. Note that the Neuron objects store the weights and the bias separately, but that the gradients you pass around should be a single vector. All of the functions you implement should work for inputs of arbitrary dimension (outputs will always be 1-dimensional).

You are welcome to add more helper functions that break these tasks down into smaller logical units. As you work through these functions, it is strongly recommended that you test incrementally using the `scratchwork` Jupyter notebook. Once you have implemented all of these functions you should be able to perform regression with a linear-activation neuron or classification with a logistic function neuron.

Testing

You have been provided with data-generating functions that produce relatively easy instances for linear regression or logistic classification, and with plotting functions for 1D regression and 2D classification. These should get you off to a good start for testing, they are not at all sufficient! In particular you should be adding other tests that come both before and after these in your implementation process. First, you should be trying out each of the functions you implement incrementally in a Jupyter notebook. Then after you've gotten things mostly working, you should be testing on harder cases with more dimensions and more widely-spread inputs, and also exploring the effect of varying the hyperparameters of your training.

Here are some (non comprehensive) ideas for what sorts of things you might want to explore:

- Both functions in `generate_data.jl` have a number of parameters you can use to make the instances harder. You can easily vary:
  - input dimension
  - data range (given by `lb` and `ub` "bounds" parameters)
  - noise (given by variance and covariance parameters)
  - number of distinct clusters (classification only)

  and you are also welcome to write your own data-generating functions to try out more complicated cases!
- All of the logging provided by `train!` is intended for you to make plots and explore options. At the very least, you should visualize the change in loss over the course of training to choose the right step size and number of iterations. But you should also investigate ways of visualizing the gradient descent path using the parameter- and/or gradient-traces.
- By transforming the inputs before training, it's possible to use single neurons to fit much more complicated models. Try augmenting a regression data set with x_i^2 and $x_i x_j$ terms to learn a quadratic function, or try transforming a classification data set into polar coordinates.

Often in machine learning, the best tool for understanding your code is data-visualization. You are strongly encouraged to do lots of plotting, and this should go well beyond the basic tools you've been provided!