MSDS621 Project 1
Using gradient descent to fit regularized linear models

Terence Parr
Data Science
University of San Francisco

Fall 2019

1 Goal

The goal of this project is to learn how to implement gradient descent function optimization, which is critical to training many important models such as neural networks. Each neuron in the network is a variant of a linear model. Because linear regression and logistic regression are important in their own right, we will use them as sample applications of gradient descent optimization. Because regularization is important to model generality, we will implement L1 and L2 regularization as well.

You will be doing your work in your repository `linreg-`*userid* cloned from github and will construct drop-in replacements for scikit-learn's models:

- `LinearRegression`
- `Ridge` (L2 regularized regression)
- `LogisticRegression` (nonregularized and L1 Lasso regularized)

(*L1 regularization with logistic regression is optional for this project.*)

2 Discussion

Most of this should be a review from your linear regression class and from the 621 lecture on gradient descent. I'm including as many of the derivations as I can because I couldn't find everything collected in one document. Also, the algebraic symbolic manipulations to move between the various equations is not obvious. There's a bunch of matrix calculus necessary to do the derivations; see The Matrix Calculus You Need For Deep Learning for more.

2.1 Linear regression

Let matrix $\mathbf{X}$ be our $n \times p$ explanatory matrix where each row, $\mathbf{x}^{(i)}$, represents an observation vector of p values, $(x_1^{(i)}, \ldots, x_p^{(i)})$ and column vector $\mathbf{y} = (y^{(1)}, \ldots, y^{(n)})$ represents the response or target variable. To make a prediction for a specific $\mathbf{x}$ observation, the linear model is:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p = \beta_0 + \sum_{i=1}^{p} \beta_i x_i$$

(One could claim β_i should be $\hat{\beta}_i$ as they are estimates, but I will be slightly less formal than that for simplicity.)

The formula is much simpler if we use vector math:

$$\hat{y} = \beta_0 + \mathbf{x} \cdot \vec{\beta} = \mathbf{x}\vec{\beta}$$

where vector $\vec{\beta}$ is column vector $(\beta_1, \ldots, \beta_p)$ and $\mathbf{x}$ is a row vector. If we augment $\mathbf{x}$ to have a "1" as the first element, calling it $\mathbf{x}'$, then we can pull β_0 into $\vec{\beta}$ and the equation simplifies to

$$\hat{y} = \mathbf{x}'\vec{\beta}$$

where vector $\vec{\beta}$ is now $(\beta_0, \beta_1, \ldots, \beta_p)$.

Similarly, if we augment the explanatory matrix to have a column of ones as the first column, calling it $\mathbf{X}'$, then we can make predictions for the entire explanatory matrix:

$$\hat{\mathbf{y}} = \mathbf{X}'\vec{\beta}$$

(From now on, I will use plain β to mean the entire vector $\vec{\beta}$.)

So, how good is the fit? Let's use the "mean squared error":

$$MSE(\beta) = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - (\mathbf{x}^{(i)} \cdot \beta))^2$$

For optimization, we don't care about scaling MSE by $\frac{1}{n}$ (though we have to be careful to adjust the learning rate), giving our "loss" function:

$$\mathscr{L}(\beta) = \sum_{i=1}^{n} (y^{(i)} - (\mathbf{x}^{(i)} \cdot \beta))^2$$

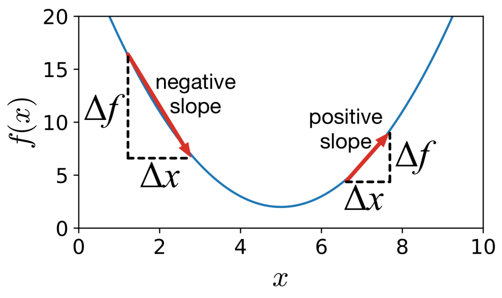
In matrix form, we get:

$$\mathscr{L}(\beta) = (\mathbf{y} - \mathbf{X}'\beta) \cdot (\mathbf{y} - \mathbf{X}'\beta)$$

Given this loss function, we know how to evaluate the quality of a β vector of coefficients. As we wiggle the coefficients around, the loss function will go up or down. So how do we know where the optimal coefficients are in $p + 1$-space? We pick an initial approximation to β and then use information about the loss function in the neighborhood of that β to decide which direction shifts it towards a better loss function:

$$\beta^{(t+1)} = \beta^{(t)} + \Delta\beta^{(t)}$$

The loss function is a quadratic (by construction), which is convex and has an exact solution. All we have to do is figure out where the loss function flattens out. Mathematically, that is when the partial derivatives of the loss function are all zero. The partial derivative for a particular dimension is just the rate of change of that dimension at a particular spot. It's like a skier examining the slope of his or her skis against the mountain one at a time. (For more, see The intuition behind gradient descent in an article that explains gradient boosting, which you will probably look at for your second machine learning course.) For an arbitrary function $f(x)$ of one variable x, here's what the loss function gradients might look like:



To go downhill, we move in the opposite direction of the (red vector) gradient. Since we are moving through coefficient space, our movement is governed by the following recurrence relation:

$$\beta^{(t+1)} = \beta^{(t)} - \nabla_{\beta}\mathscr{L}(\beta^{(t)})$$

We're done when we find a β such that:

$\nabla_\beta \mathscr{L}(\beta) = \vec{0}$

Instead of solving that equation for β symbolically, we'll use gradient descent to minimize the loss function. It's important to learn about this numerical technique because there are lots and lots of loss functions that are not simple little quadratics with symbolic solutions. When we add regularization and move to logistic regression, we will need this gradient descent method.

The derivative of our loss function with respect to β is:

$$\nabla_\beta \mathscr{L}(\beta) = -2\mathbf{X}'^T(\mathbf{y} - \mathbf{X}'\beta)$$

which gives us a $p+1$-sized vector of partial derivatives, one for each β_i including β_0. That means that the update equation is:

$$\beta^{(t+1)} = \beta^{(t)} - \eta\mathbf{X}'^T(\mathbf{y} - \mathbf{X}'\beta^{(t)})$$

where η is a scalar "learning rate" that controls the magnitude of the step we take at each time t and we have folded the 2 into η.

2.2 Gradient-descent algorithm

The core minimization algorithm for the mean squared error loss function is:

```
Algorithm: basic_minimize($\mathbf{X}$, $\mathbf{y}$, $\mathscr{L}$, $\eta$) returns coefficients $\vec{b}$
Let $\vec{b} \sim 2N(0,1) - 1$     (init b with random $p+1$-sized vector with elements in [-1,1))
$\mathbf{X}' = (\vec{\mathbf{1}}, \mathbf{X})$               (Add first column of 1s to data)
repeat
  $\nabla_{\vec{b}} = -\mathbf{X}'^T(\mathbf{y} - \mathbf{X}'\vec{b})$
  $\vec{b} = \vec{b} - \eta\nabla_{\vec{b}}$
until $|(\mathscr{L}(\vec{b}) - \mathscr{L}(\vec{b}_{prev}))| < precision$;
return $\vec{b}$
```

We can make a lot more progress towards our goal if we add **momentum** to the particle traversing the loss function surface. The basic gradient descent algorithm has trouble when it's stuck in a valley where one direction is shallow and the other is steep. If the particle keeps moving in the same direction, it will start to accelerate because the vectors keep adding in that direction. It is surprisingly easy to update the algorithm to use momentum:

```
Algorithm: minimize($\mathbf{X}$, $\mathbf{y}$, $\mathscr{L}$, $\eta$, $\gamma$) returns coefficients $\vec{b}$
Let $\vec{b} \sim 2N(0,1) - 1$    ($random\ p+1$-sized vector with elements in [-1,1))
$\mathbf{X}' = (\vec{\mathbf{1}}, \mathbf{X})$    (Add first column of 1s)
repeat
    $\nabla_{\vec{b}} = -\mathbf{X}'^T(\mathbf{y} - \mathbf{X}'\vec{b})$
    $\vec{v} = \gamma\vec{v} + \eta\nabla_{\vec{b}}$
    $\vec{b} = \vec{b} - \vec{v}$
until $|(\mathscr{L}(\vec{b}) - \mathscr{L}(\vec{b}_{prev}))| < precision$;
return $\vec{b}$
```

Another thing we need to do for speed is to **normalize** all of the explanatory variables in **X** because we have a single global learning rate, rather than a different learning rate per $p+1$ dimensions. If x_1 is in the range [0,100] and x_2 is in the range [0,0.001], the learning rate that is appropriate for x_2 (so we converge) is way too slow for x_1. Normalization brings them all into the same range. For each x_i value, we subtract the mean and divide by the standard deviation to get standard values. This is particularly important for L2 Ridge regression because the solution relies on mean-centered x_i.

This algorithm is pretty efficient but to really go much faster we need a different learning rate per dimension. You can learn more here:

1. An overview of gradient descent optimization algorithms
2. CS231n Convolutional Neural Networks for Visual Recognition
3. AdaDelta: An adaptive learning rate method

Adding per-dimension learning rate adjustments is easy using Adagrad. It's the next step on the road to even more sophisticated mechanisms that are needed for deep learning neural networks. With Adagrad, I found that momentum doesn't help much so we can discard it. Here's the Adagrad algorithm that you should use for your projects:

```
Algorithm: adagrad_minimize($\mathbf{X}$, $\mathbf{y}$, $\mathscr{L}$, $\eta$, $\epsilon$=1e-5) returns coefficients $\vec{b}$
Let $\vec{b} \sim 2N(0,1) - 1$    ($random\ p+1$-sized vector with elements in [-1,1))
$h = \vec{0}$    ($p+1$-sized sum of squared gradient history)
$\mathbf{X}' = (\vec{\mathbf{1}}, \mathbf{X})$    (Add first column of 1s)
repeat
    $\nabla_{\vec{b}} = -\mathbf{X}'^T(\mathbf{y} - \mathbf{X}'\vec{b})$
    $h \mathrel{+}= \nabla_{\vec{b}} \cdot \nabla_{\vec{b}}$
    $\vec{b} = \vec{b} - \eta * \frac{\nabla_{\vec{b}}}{(\sqrt{h}+\epsilon)}$
until $||(\mathscr{L}(\vec{b}) - \mathscr{L}(\vec{b}_{prev}))||_2 < precision$;
return $\vec{b}$
```

You're going to build a generic version of this that will be useful for all variations of linear and logistic regression. The only difference is that we pass in the loss function and its gradient:

```
Algorithm: adagrad_minimize($\mathbf{X}$, $\mathbf{y}$, $\mathscr{L}$, $\nabla\mathscr{L}$, $\eta$, $\epsilon$=1e-5) returns coefficients $\vec{b}$
Let $\vec{b} \sim 2N(0,1)-1$     (random $p+1$-sized vector with elements in [-1,1))
$h = \vec{0}$                    ($p+1$-sized sum of squared gradient history)
$\mathbf{X}' = (\vec{\mathbf{1}}, \mathbf{X})$              (Add first column of 1s)
repeat
    $h \mathrel{+}= \nabla\mathscr{L} \cdot \nabla\mathscr{L}$
    $\vec{b} = \vec{b} - \eta * \frac{\nabla\mathscr{L}}{(\sqrt{h}+\epsilon)}$
until $|(\mathscr{L}(\vec{b}) - \mathscr{L}(\vec{b}_{prev}))| < precision$;
return $\vec{b}$
```

2.3 L2 ridge regularization for regression

Regularization is primarily a means to improve the generality of our fitted model, but it can also be used to shrink the coefficient for useless features down towards to or actually to zero. It also makes training less fussy about codependent features in the explanatory matrix. The idea is very simple: let's penalize the loss function when coefficients get too big. Here is the loss function that incorporates L2 (Ridge) regularization:

$$\mathscr{L}(\beta) = (\mathbf{y} - \mathbf{X}\beta) \cdot (\mathbf{y} - \mathbf{X}\beta) + \lambda\beta \cdot \beta$$

where $\beta \cdot \beta$ is the sum of the squared coefficients. (See ESLII *The elements of statistical learning* p63 in Hastie et al.) In this case, β does *not* include the y-intercept, β_0. The slope is what we care about getting right and that slope could be paired with lots of different y-intercepts. We don't want to penalize the fit based upon simple shifts up and down of the entire line. Instead, users must ensure that the incoming x_i variables are mean centered, which we do as part of normalization. Also there is no left-edge column of ones in the first column of $\mathbf{X}$.

In order to use gradient descent, we need the gradient of the loss function. The gradient for the non-regularized case was:

$$\nabla_\beta \mathscr{L}(\beta) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta)$$

but we've added the $\lambda\beta \cdot \beta$ term and need to incorporate its gradient:

$$\nabla_\beta \mathscr{L}(\beta) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) + 2\lambda\beta$$

Dropping the constants again, we get update equation:

$$\beta^{(t+1)} = \beta^{(t)} - \eta \mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) + \lambda\beta$$

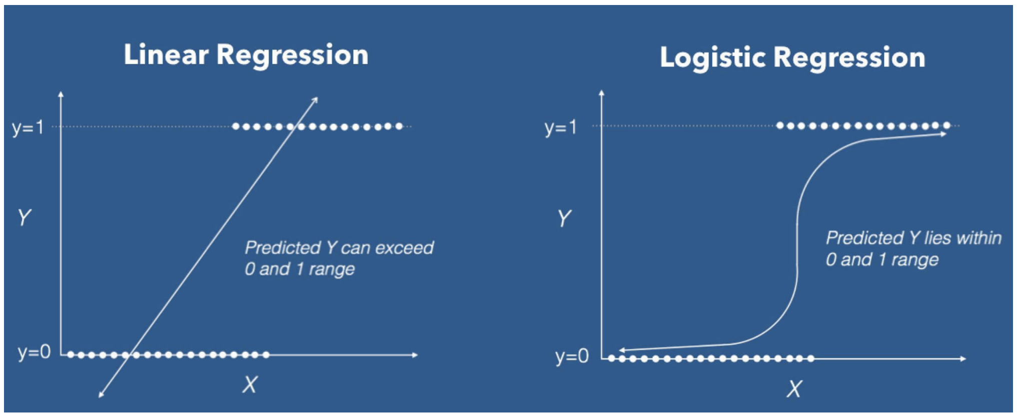
The algorithm is almost identical as that for regular regression, except that we don't add the column of one's to $\mathbf{X}$, don't put β_0 into β, and return the mean of $\mathbf{y}$ as β_0. In your real project code, there is a parameter, `addB0`, that dictates whether or not to use augmented vectors and matrices. We pass the same loss function, MSE, to the minimizer but now we pass loss gradient:

$$\nabla_\beta \mathscr{L}(\beta) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta)$$

2.4 Logistic regression

Linear regression makes numerical predictions, and these predictions are typically continuous values not integers. If we want to build a classifier, however, we need some kind of model that predicts discrete classes, such as 0 and 1 (for a two-class problem). We could use a linear regression model but a line predicts continuous values below, in between, and above range [0,1]. A classifier should give only two values 0 or 1.

What we can do is apply a function to the linear regression output that forces the value to a probability (between 0 and 1) that the incoming $\mathbf{x}$ feature vector should be classified as class 1. In a sense, this modified output would give its confidence that $\mathbf{x}$ is a 1. The following image nicely summarizes the difference between linear and logistic regression.[1]



To make a classifier out of such a model, we can simply threshold the probability so that the classifier predicts class 0 when the probability is < 0.5 and predicts class 1 when the probability is ≥ 0.5.

The function we apply is a sigmoid called the logistic function, hence, logistic regression:

[1]Image credit: Logistic Regression – A Complete Tutorial With Examples in R

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{1+e^z}$$

It is technically still a regressor not a classifier; it only becomes a classifier when we apply a threshold to the resulting probability.

If we apply the sigmoid function to the output of a linear model, we get:

$$p(\mathbf{x}') = \sigma(\mathbf{x}'\beta) = \frac{1}{1+e^{-\mathbf{x}'\beta}}$$

where $p(\mathbf{x}')$ is the probability that $y = 1$ and $\mathbf{x}'$ is the 1-augmented $\mathbf{x}$ feature row vector.

2.4.1 Why the logistic regression equation gives log odds

Before moving on to fitting a logistic regression model, let's move backwards from this simple sigmoid applied to a linear model towards the "log odd" description textbooks normally start with. The goal is to get a simple linear model, as opposed to the sigmoid, on the right-hand side and then ask what the left-hand side describes. Well, the sigmoid generates a probability that looks pretty complicated, but if we represent that as odds instead things start to cancel (which gets us closer to our goal of isolating the linear equation on the right-hand side). Recall that $odds = \frac{p}{1-p}$, which gives us:

$$odds = \frac{\frac{1}{1+e^{-\mathbf{x}'\beta}}}{1 - \frac{1}{1+e^{-\mathbf{x}'\beta}}}$$

Substituting $\frac{1+e^{-\mathbf{x}'\beta}}{1+e^{-\mathbf{x}'\beta}}$ for 1 in denominator to combine terms, we get:

$$odds = \frac{\frac{1}{1+e^{-\mathbf{x}'\beta}}}{\frac{1+e^{-\mathbf{x}'\beta}-1}{1+e^{-\mathbf{x}'\beta}}}$$

$$odds = \frac{\frac{1}{1+e^{-\mathbf{x}'\beta}}}{\frac{e^{-\mathbf{x}'\beta}}{1+e^{-\mathbf{x}'\beta}}}$$

$$odds = \frac{1+e^{-\mathbf{x}'\beta}}{(1+e^{-\mathbf{x}'\beta})(e^{-\mathbf{x}'\beta})}$$

$$odds = \frac{1}{e^{-\mathbf{x}'\beta}} = e^{\mathbf{x}'\beta}$$

Now, take the log of both sides:

$$log(odds) = \mathbf{x}'\beta$$

and that is why the linear regression equation gives log odds in the logistic case.

2.4.2 Optimizing the logistic regression equation

To estimate β for logistic regression, we need a loss function that indicates how good the fit is. We use a "maximum likelihood" measure for a loss function. For the two-class case, the likelihood of the sigmoid derived from some β fitting the $(\mathbf{X}, \mathbf{y})$ data set is:

$$\mathscr{L}(\beta) = \prod_{i=1}^{n} \begin{cases} p(\mathbf{x}'^{(i)}; \beta) & \text{if } y^{(i)} = 1 \\ 1 - p(\mathbf{x}'^{(i)}; \beta) & \text{if } y^{(i)} = 0 \end{cases}$$

Since log is monotonic and doesn't affect optimization via the loss function, we can use it to flip the product to a summation:

$$\mathscr{L}(\beta) = \sum_{i=1}^{n} \begin{cases} log(p(\mathbf{x}'^{(i)}; \beta)) & \text{if } y^{(i)} = 1 \\ log(1 - p(\mathbf{x}'^{(i)}; \beta)) & \text{if } y^{(i)} = 0 \end{cases}$$

Using $y^{(i)}$ and $(1-y^{(i)})$ terms to gate the two log terms in and out let us remove the choice operator:

$$\mathscr{L}(\beta) = \sum_{i=1}^{n} \left\{ y^{(i)} log(p(\mathbf{x}'^{(i)}; \beta)) + (1 - y^{(i)}) log(1 - p(\mathbf{x}'^{(i)}; \beta)) \right\}$$

Recall from above the derivation from $p(\mathbf{x})$ to isolate $\mathbf{x}'\beta$ on the right-hand side, which led to the log-odds on the left-hand side. Now we have log-probability and, if we flip the probability to odds, we end up with $e^{\mathbf{x}'^{(i)}\beta}$ inside the log which just gives us $\mathbf{x}'^{(i)}\beta$ for the first term. Ultimately, with more substitutions, we can reduce the maximum likelihood to equation 4.20 from ESLII p120:

$$\mathscr{L}(\beta) = \sum_{i=1}^{n} \left\{ y^{(i)} \mathbf{x}'^{(i)} \beta - log(1 + e^{\mathbf{x}'\beta}) \right\}$$

The gradient derivation w.r.t. β:

$$
\begin{array}{rcl}
\nabla_\beta \mathscr{L}(\beta) = \frac{\partial \mathscr{L}(\beta)}{\partial \beta} & = & \sum_{i=1}^{n} \{ \frac{\partial}{\partial\beta} y^{(i)} \mathbf{x}'^{(i)} \beta - \frac{\partial}{\partial\beta} log(1 + e^{\mathbf{x}'^{(i)}\beta}) \} \\
& = & \sum_{i=1}^{n} \{ y^{(i)} \mathbf{x}'^{(i)} - \frac{\partial}{\partial\beta} log(1 + e^{\mathbf{x}'^{(i)}\beta}) \} \\
& = & \sum_{i=1}^{n} \{ y^{(i)} \mathbf{x}'^{(i)} - \frac{1}{log(1+e^{\mathbf{x}'^{(i)}\beta})} \frac{\partial}{\partial\beta} (1 + e^{\mathbf{x}'^{(i)}\beta}) \} \\
& = & \sum_{i=1}^{n} \{ y^{(i)} \mathbf{x}'^{(i)} - \frac{1}{log(1+e^{\mathbf{x}'^{(i)}\beta})} \frac{\partial}{\partial\beta} e^{\mathbf{x}'^{(i)}\beta} \} \\
& = & \sum_{i=1}^{n} \{ y^{(i)} \mathbf{x}'^{(i)} - \frac{1}{log(1+e^{\mathbf{x}'^{(i)}\beta})} e^{\mathbf{x}'^{(i)}\beta} \frac{\partial}{\partial\beta} \mathbf{x}'^{(i)}\beta \} \\
& = & \sum_{i=1}^{n} \{ y^{(i)} \mathbf{x}'^{(i)} - \frac{1}{1+e^{\mathbf{x}'^{(i)}\beta}} \mathbf{x}'^{(i)} e^{\mathbf{x}'^{(i)}\beta} \} \\
& = & \sum_{i=1}^{n} \{ y^{(i)} \mathbf{x}'^{(i)} - \mathbf{x}'^{(i)} \frac{e^{\mathbf{x}'^{(i)}\beta}}{1+e^{\mathbf{x}'^{(i)}\beta}} \} \\
& = & \sum_{i=1}^{n} \{ y^{(i)} \mathbf{x}'^{(i)} - \mathbf{x}'^{(i)} p(\mathbf{x}'^{(i)};\beta) \} \\
& = & \sum_{i=1}^{n} \{ \mathbf{x}'^{(i)} (y^{(i)} - p(\mathbf{x}'^{(i)};\beta)) \} \\
& = & -\mathbf{X}^T(\mathbf{y} - \sigma(\mathbf{X}\cdot\beta))
\end{array}
$$

Now that we have $\nabla_\beta \mathscr{L}(\beta)$, we can reuse the minimization function above to obtain β coefficients for logistic regression. (That equation is the negation of equation 4.21 from ESLII p 120; they drop the negation because they're setting equal to zero.) Pass in the log likelihood as the loss function and this log likelihood gradient as the loss gradient function.

2.4.3 L1 Lasso regularization

As a resource, see Andrew Ng's Machine Learning Course in Python (Regularized Logistic Regression) + Lasso Regression. I noticed a few errors maybe, btw, in this person's implementations.

Just as we did for linear regression, we can add regularization to logistic regression to constrain the β coefficients. We're going to do L1 (Lasso) regularization, which has the convenient property that it can send useless coefficients to zero. Here is the loss function that we need to minimize:

$$\mathscr{L}(\beta) = \frac{1}{n} \left\{ \sum_{i=1}^{n} \left\{ y^{(i)} \mathbf{x}'^{(i)} \beta - log(1 + e^{\mathbf{x}'\beta}) \right\} - \lambda \sum_{j=1}^{p} |\beta| \right\}$$

L2 uses the squared magnitude, but L1 uses the sum of the magnitude. Note that β_0 is not included in the regularization term to avoid penalizing for the magnitude of the y-intercept. The tricky bit to computing the gradient of this loss function is that we need to solve for β_0 coefficient differently than for the other p coefficients. To make matters worse, the computation for $\beta_1..\beta_p$ involves β_0 and vice versa. For β_0, the gradient is:

$$\nabla_{\beta_0} \mathscr{L}(\beta) = \frac{1}{n} \sum_{i=1}^{n} \{ {x_0}^{(i)} (y^{(i)} - p(\mathbf{x}'^{(i)};\beta))$$

The $\frac{1}{n}$ scales the gradient by the size of the data set so that different sized training sets are normalized to use the same gradient. The ESLII book does not include this term as it's not necessary for optimization purposes (it just raises the loss surface up or down without changing its shape). I noticed that it was necessary to include this term to matchup coefficients computed by sklearn, so let's keep it in. Andrew Ng's loss function and loss function gradient, which is like what I'm doing, are shown in this stack overflow post. In vector notation, that gradient is:

$$\nabla_{\beta_0}\mathscr{L}(\beta) = \frac{1}{n}x'_0(\mathbf{y} - \sigma(\mathbf{X}' \cdot \beta))$$

but x'_0 is just $\vec{\mathbf{1}}$:

$$\nabla_{\beta_0}\mathscr{L}(\beta) = \frac{1}{n}\vec{\mathbf{1}} \cdot (\mathbf{y} - \sigma(\mathbf{X}' \cdot \beta))$$

which just sums up the error term in the parentheses and divides by n. That's otherwise known as the average of the error term:

$$\nabla_{\beta_0}\mathscr{L}(\beta) = mean(\mathbf{y} - \sigma(\mathbf{X}' \cdot \beta))$$

For $\beta_1..\beta_p$, we need to carefully identify what parts of $\mathbf{X}'$ and β we are using. To get the error term, we use the full augmented-with-ones matrix $\mathbf{X}'$ and β: $\mathbf{y} - \sigma(\mathbf{X}' \cdot \beta)$. From then on, we use just $1..p$ which means not using the augmented matrix:

$$\nabla_{\beta^{(1..p)}}\mathscr{L}(\beta) = \frac{1}{n}\left\{\mathbf{X}^T(\mathbf{y} - \sigma(\mathbf{X}' \cdot \beta)) - \lambda\,\mathrm{sign}(\beta^{(1..p)})\right\}$$

Then the β_0 gradient and these gradients are combined to get a full ∇_β.

3 Deliverables

You must implement the following functions in `linreg.py` in the root directory of your project repo (linreg.py starter kit):

- `MSE(X,y,B,lmbda)`
$$\mathscr{L}(\beta) = (\mathbf{y} - \mathbf{X}'\beta) \cdot (\mathbf{y} - \mathbf{X}'\beta)$$
- `loss_gradient(X, y, B, lmbda)`
$$\nabla_\beta\,\mathscr{L}(\beta) = -2\mathbf{X}'^T(\mathbf{y} - \mathbf{X}'\beta)$$

- `loss_ridge(X, y, B, lmbda)`
$$\mathscr{L}(\beta) = (\mathbf{y} - \mathbf{X}\beta) \cdot (\mathbf{y} - \mathbf{X}\beta) + \lambda\beta \cdot \beta$$
- `loss_gradient_ridge(X, y, B, lmbda)`
$$\nabla_\beta \mathscr{L}(\beta) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) + 2\lambda\beta$$
- `sigmoid(z)`
- `log_likelihood(X, y, B)`
$$\mathscr{L}(\beta) = \sum_{i=1}^{n} \left\{ y^{(i)}\mathbf{x}'^{(i)}\beta - log(1 + e^{\mathbf{x}'\beta}) \right\}$$
- `log_likelihood_gradient(X, y, B, lmbda)`
$$\nabla_\beta \mathscr{L}(\beta) = -\mathbf{X}^T(\mathbf{y} - \sigma(\mathbf{X} \cdot \beta))$$
- `minimizer(X, y, loss, loss_gradient, eta, ...)`

Notice that we are passing a λ parameter for consistency to all loss and loss gradient functions. That way the same minimizer function can work with all of them. The only difference between the various regression techniques is the loss and loss function pass to the minimizer.

Next, you have to implement two classes that mimic how sklearn's models work:

- class `RidgeRegression621`
- class `LogisticRegression621`

class `LinearRegression621` is provided for you as a template:

```
class LinearRegression621:
    def __init__(self, eta=0.00001, lmbda=0.0, max_iter=1000):
        self.eta = eta
        self.lmbda = lmbda
        self.max_iter = max_iter

    def predict(self, X):
        n = X.shape[0]
        B0 = np.ones(shape=(n, 1))
        X = np.hstack([B0, X])
        return np.dot(X, self.B)

    def fit(self, X, y):
        self.B = minimizer(X, y,
                           MSE, loss_gradient,
                           self.eta, self.lmbda, self.max_iter)
```

You can mostly cut-and-paste from that into those other two. The `predict()` and `fit()` methods should be familiar to you.

4 Extensions

If you would like to see how regularization works with logistic regression, implement functions `L1_log_likelihood_gradient(X, y, B, lmbda)` and `def L1_log_likelihood(X, y, B, lmbda)` plus create class `LassoLogistic621`, all in the same `linreg.py` file. You can uncomment functions `test_lasso_iris()`, `test_lasso_wine`, and `test_lasso_synthetic` in `test_class.py` to test your L1 Lasso code.

5 Evaluation

Your project will be evaluated using these predefined test rigs: test_regr.py and test_class.py. Please do not modify these as I will copy fresh versions into your repos when testing.

Here is the output I get on my solution, including the optional L1 lasso tests, but you are free to ignore that extra part of the work if you like. Because you are given complete unit tests *a priori*, I will be grading in a binary fashion as usual. For full credit, you must get all 9 (nonoptional) unit tests passing. Each failed test cost to 10 points from 100 total points. I suggest that you run your tests at Amazon so that you can verify your code works on a different computer, otherwise you will be very sad with a zero grade. There is no such thing as partial credit for late projects; late projects get zero.

```
$ python -m pytest -v test_regr.py
=================================== test session starts ====================================
platform darwin -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0 -- ...
cachedir: .pytest_cache
rootdir: /Users/parrt/courses/msds621-private/projects/regularization, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, doctestplus-0.2.0, arraydiff-0.3
collected 6 items

test_regr.py::test_synthetic PASSED                                                  [ 16%]
test_regr.py::test_ridge_synthetic PASSED                                            [ 33%]
test_regr.py::test_boston PASSED                                                     [ 50%]
test_regr.py::test_boston_noise PASSED                                               [ 66%]
test_regr.py::test_ridge_boston PASSED                                               [ 83%]
test_regr.py::test_ridge_boston_noise PASSED                                         [100%]

================================= 6 passed in 1.18 seconds =================================
```

```
$ python -m pytest -v test_class.py
=================================== test session starts ====================================
platform darwin -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0 -- ...
cachedir: .pytest_cache
rootdir: /Users/parrt/courses/msds621-private/projects/regularization, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, doctestplus-0.2.0, arraydiff-0.3
collected 6 items

test_class.py::test_synthetic PASSED                                                 [ 16%]
test_class.py::test_lasso_synthetic PASSED                                           [ 33%]
test_class.py::test_wine PASSED                                                  [ 50%]
test_class.py::test_lasso_wine PASSED                                            [ 66%]
test_class.py::test_iris PASSED                                                  [ 83%]
test_class.py::test_lasso_iris PASSED                                            [100%]

===================================== warnings summary =====================================
test_class.py::test_wine
  logistic.py:433: FutureWarning: Default solver will be changed to 'lbfgs' in 0.22. Specify a solver to silence th
    FutureWarning)

-- Docs: https://docs.pytest.org/en/latest/warnings.html
========================== 6 passed, 1 warnings in 7.02 seconds ===========================
```

You can ignore that warning about lbfgs.

6 Submission

To submit your project, ensure that `linreg.py` is submitted to your repository. That file must be in the root directory of your `linreg-`*userid* repository.