MSDS621 Project 1
Using gradient descent to fit regularized linear models

Terence Parr
Data Science
University of San Francisco

Fall 2019

1   Goal

The goals of this project are *(i)* to learn how to implement gradient descent function optimization and *(ii)* to lock in your understanding of regularized linear models. Gradient descent optimizers are critical to training many important models, such as neural networks where each neuron in the network is a variant of a linear model. Because regularization is important to model generality, we will implement L2 regularization with an option to do L1 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 only)

L1 regularization with logistic regression is optional for this project.

2   Discussion

Training linear models involves optimizing a function, called a loss function, that describes how well the model's predictions match the known target values; i.e., how well the $\hat{y}$ values match y for all $y^{(i)}$ in the $(\mathbf{X}, \mathbf{y})$ training set. Given the model parameters (β coefficients), the loss function gives a scalar value indicating a "goodness of fit" for the training data. The smaller the loss, the better the fit to the training data. Optimizing the loss function means finding the model parameters

that give the minimum loss function scalar value. For the vanilla linear regression model, the loss function is just the mean squared error (MSE):

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

or, in vector notation:

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

where vector $\vec{\beta}$ is column vector $(\beta_0, \beta_1, \ldots, \beta_p)$ and $\mathbf{x}'$ is augmented row vector $(1, x_1, \ldots, x_p)$ and $\hat{y} = \mathbf{x}'\vec{\beta}$. Without regularization, the loss function is a quadratic (by construction), which is convex, and has an exact solution. But, we are going to solve everything iteratively rather than symbolically as it's necessary for logistic regression and sometimes for regularized linear regression. (I'll drop the vector from $\vec{\beta}$ now for convenience.)

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:

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

where $\Delta\beta^{(t)}$ is a shift in the direction of lower loss function value. That $\Delta\beta^{(t)}$ turns out to be the opposite of the gradient vector, as shown in this 1D case:
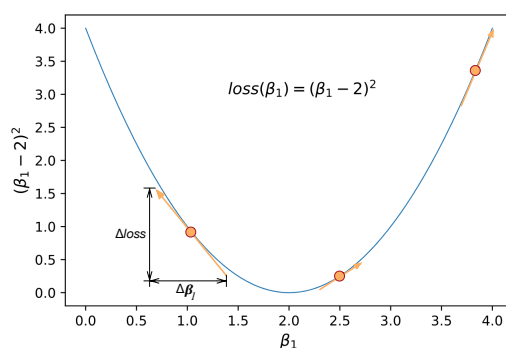
Here's what a regression model loss function looks like for (β_0, β_1):
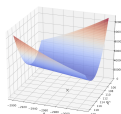
The gradient vector is the vector of slopes in each of $p+1$ directions and the slope always points in the up direction, so we want to go in the negative of the gradient to go downhill. Our movement is governed by the following recurrence relation:

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

where η is a scalar "learning rate" that controls the magnitude of the step we take at each step t. We're done optimizing when we find a β where all slopes are zero/flat:

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

So, in a nutshell, to optimize a loss function we need the loss function and the loss of function gradient. Don't worry about dusting off your calculus knowlege—I'm providing all the symbolic gradients for you in this document. For example, here is a complete Python program for $p = 1$ that optimizes $y = (x - 2)^2$ using the derivative $\frac{dy}{dx} = 2(x - 2)$:

```
import numpy as np

def f(b) : return (b-2)**2
def gradient(b): return 2*(b-2)

def minimize(f, rate=.3, niter=7):
    b = np.random.random()   # initial guess at optimal b
    for i in range(niter):
        print(f"i:02d: beta_1=b:.2f, f(beta_1)=f(b):.2f, gradient gradient(b):.2f")
        b = b - rate * gradient(b)
    return b

minimize(f)
```

The program generates the following output.

```
00: beta_1=0.34, f(beta_1)=2.77, gradient -3.33
01: beta_1=1.33, f(beta_1)=0.44, gradient -1.33
02: beta_1=1.73, f(beta_1)=0.07, gradient -0.53
03: beta_1=1.89, f(beta_1)=0.01, gradient -0.21
04: beta_1=1.96, f(beta_1)=0.00, gradient -0.09
05: beta_1=1.98, f(beta_1)=0.00, gradient -0.03
06: beta_1=1.99, f(beta_1)=0.00, gradient -0.01
```

This program simply terminates after a certain number of iterations, but in general we have to continue until the gradient is zero or β doesn't move from one step to the next.

3

Once we have a suitable `minimize()` function, we can use it to optimize linear and logistic regression equations, both normal and regularized. Here are the loss function and gradient of the loss function needed to train a logistic model using `minimize()`:

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

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

Per the appendix, we're going to use a general minimizer function that optimizes $\mathscr{L}$ by chasing $-\nabla \mathscr{L}$ and using a separate learning rate per $p + 1$ dimensions:

Algorithm: *adagrad_minimize*($\mathbf{X}$, $\mathbf{y}$, $\mathscr{L}$, $\nabla \mathscr{L}$, η, ϵ=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
    $\vec{h} \mathrel{+}= \nabla \mathscr{L} \otimes \nabla \mathscr{L}$ (*track sum of squared partials, use element-wise product*)
    $\vec{b} = \vec{b} - \eta * \frac{\nabla \mathscr{L}}{(\sqrt{\vec{h}} + \epsilon)}$
until $|(\mathscr{L}(\vec{b}) - \mathscr{L}(\vec{b}_{prev}))| < precision$;
return $\vec{b}$

In your implementation, you'll need a parameter, `addB0`, that dictates whether or not to use augmented vectors and matrices. We always set `addB0=True` except for Ridge linear regression, which computes β_0 outside of the minimization process (as just $\overline{y}$) assuming standardized variables.

As part of this project, 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:

```python
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 = minimize(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.

3   Getting started

I recommend that you play around with the 1D $y = (x - 2)^2$ example above and make sure you fully understand how it works. This will make it easier to expand into $p + 1$ space. Also walk through the Notebook on visualizing gradient descent.

For more on gradient descent, you can look at The intuition behind gradient descent in an article that explains gradient boosting, which you will probably look at for your second machine learning course.

To learn more than you ever wanted to learn about gradients, check out The Matrix Calculus You Need For Deep Learning.

To actually begin the project, create `linreg.py` in your repo and define just the functions you will need to get basic linear regression working: `minimize()`, `MSE()`, and `loss_gradient()`. Try to debug your code using a single x_1 vs y regression. If that works, then try with $\mathbf{x} = (x_1, x_2)$. If it fails, then you know in how you handle multiple dimensions, rather than the core of your optimization loop. To aid in your debugging, you can check out some of the visualization stuff I did in the visualization notebook.

Once you have your minimizer working, add the `LinearyRegression621` class and then try to get the unit test working. Please note that the starter kit has a number of defined functions in the classes that are used by the unit tests. Please follow that API.

This project looks complicated because of all the math notation, but all you are really doing is converting math notation into Python numpy operations. If you don't use numpy vector operations (like dot product), your code will be too slow and will get a grade penalty.

4   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)`
$$\nabla_\beta \mathscr{L}(\beta) = -\mathbf{X}'^T(\mathbf{y} - \mathbf{X}'\beta)$$

- `loss_ridge(X, y, B, lmbda)` for $\beta_{1..p}$ ($\beta_0 = \overline{\mathbf{y}}$)
$$\mathscr{L}(\beta) = (\mathbf{y} - \mathbf{X}\beta) \cdot (\mathbf{y} - \mathbf{X}\beta) + \lambda\beta \cdot \beta$$

- `loss_gradient_ridge(X, y, B, lmbda)` for $\beta_{1..p}$
$$\nabla_\beta \mathscr{L}(\beta) = -\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) + \lambda\beta$$

- `sigmoid(z)`
- `log_likelihood(X, y, B)` for $\beta_{0..p}$ (return negative of max likelihood to make it a loss function)
$$\mathscr{L}(\beta) = -\sum_{i=1}^{n} \left\{ y^{(i)}\mathbf{x}'^{(i)}\beta - log(1 + e^{\mathbf{x}'^{(i)}\beta}) \right\}$$

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

- `minimize(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 minimize function can work with all of them. The only difference between the various regression techniques is the loss and loss function passed to the minimize function.

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

- class `RidgeRegression621`
- class `LogisticRegression621`

where class `LinearRegression621` is provided for you in the starter kit.

5 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 use the `test_class_lasso.py` script to test your L1 Lasso code. This part is optional and won't be graded.

```
$ python -m pytest -v test_class_lasso.py
=================================== test session starts ===================================
platform darwin -- Python 3.7.3, pytest-4.0.2, py-1.7.0, pluggy-0.8.0 -- /Users/parrt/anaconda3/bin/python
cachedir: .pytest_cache
rootdir: /Users/parrt/courses/msds621-private/projects/linreg, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, doctestplus-0.2.0, arraydiff-0.3
collected 3 items

test_class_lasso.py::test_lasso_synthetic PASSED                                   [ 33%]
test_class_lasso.py::test_lasso_wine PASSED                                        [ 66%]
test_class_lasso.py::test_lasso_iris PASSED                                        [100%]


================================= 3 passed in 6.22 seconds =================================
```

6   Evaluation

Your project will be evaluated using these predefined test rigs: test_regr.py and test_class.py.
File test_class_lasso.py is for your own use and is optional; it is testing the optional L1 lasso for
classification part of the project. Please do not modify these as I will copy fresh versions into your
repos when testing.

Here is the output I get from my solution.

```
$ python -m pytest -v test_regr.py
============================ test session starts ============================
platform darwin -- Python 3.8.6, pytest-6.2.1, py-1.10.0, pluggy-0.13.1
cachedir: .pytest_cache
rootdir: /Users/parrt/courses/msds621-private/projects/linreg
plugins: anyio-2.0.2
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 1.46s ============================


$ python -m pytest -v test_class.py
============================ test session starts ============================
platform darwin -- Python 3.8.6, pytest-6.2.1, py-1.10.0, pluggy-0.13.1
cachedir: .pytest_cache
rootdir: /Users/parrt/courses/msds621-private/projects/linreg
plugins: anyio-2.0.2
collected 3 items

test_class.py::test_synthetic PASSED                            [ 33%]
test_class.py::test_wine PASSED                                 [ 66%]
test_class.py::test_iris PASSED                                 [100%]
```

```
============================== 3 passed in 5.18s ==============================
```

You can ignore that `FutureWarning` about `lbfgs`.

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.

If you do not use vector operations with NumPy, the project will be to slow. Your unit tests must finish in less than 10 seconds to get full credit. Failure to terminate fast enough costs you 15%.

7 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.

Appendices

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.

A 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:

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

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 $\vec{\beta}$ and assume that $\vec{\beta}$ includes β_0.)

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. 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:

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

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

A.1   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 usual loss function that incorporates an L2 (Ridge) regularization (Lagrange multiplier) penalty term:

$$\mathscr{L}(\beta, \lambda) = \sum_{i=1}^{n} \left\{ [y^{(i)} - (\beta_0 + \sum_{j=1}^{p} x_j^{(i)} \beta_j)]^2 \right\} + \lambda \sum_{j=1}^{p} \beta_j^2$$

where it's best to be explicit with summations rather than use vector notation because β_0 is going to pop in and out of our equations below. (See ESLII *The elements of statistical learning* p63 in Hastie et al.) The penalty term 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. All we care about is the size of the coefficients.

In order to use gradient descent, we need the gradient of the loss function. Because the terms are additive, we can compute their derivatives separately. Also, if we mean-center the $\mathbf{x}$ feature vectors, we can take β_0 out of our equations and estimate it with the average $\mathbf{y}$ value. Removing β_0 makes the notation much simpler because we don't have to distinguish between vector β with and without β_0:

The derivation of the gradient for the *non-regularized case* with respect to some β_k for $k \geq 1$ is:

$$
\begin{aligned}
\frac{\partial}{\partial \beta_k} \mathcal{L}(\beta, \lambda) &= \sum_{i=1}^{n} 2[y^{(i)} - \sum_{j=1}^{p} x_j^{(i)} \beta_j)] \frac{\partial}{\partial \beta_k} [y^{(i)} - \sum_{j=1}^{p} x_j^{(i)} \beta_j)] \\
&= 2 \sum_{i=1}^{n} [y^{(i)} - \sum_{j=1}^{p} x_j^{(i)} \beta_j)] \frac{\partial}{\partial \beta_k} [-x_k^{(i)}] \\
&= -2 \sum_{i=1}^{n} x_k^{(i)} [y^{(i)} - \sum_{j=1}^{p} x_j^{(i)} \beta_j)]
\end{aligned}
$$

In vector notation, that gives us (non-augmented-with-one $\mathbf{X}$):

$$
\nabla_\beta \mathcal{L}(\beta) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) \text{ for } \beta_{k \geq 1}
$$

The derivative of the penalty term (w/o β_0) is:

$$
\frac{\partial}{\partial \beta_k} \lambda \sum_{j=1}^{p} \beta_j^2 = 2\lambda \beta_k \text{ for } \beta_{k \geq 1}
$$

In vector notation, we get the following combined L2 regularization gradient:

$$
\nabla_\beta \mathcal{L}(\beta, \lambda) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta') + 2\lambda \beta \text{ for } \beta_{k \geq 1}
$$

Dropping the constants again, but not the sign, 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 non-regularized 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 a similar loss function, MSE plus the L2 penalty term, to the minimizer and must pass the loss gradient that contains the derivative of the L2 penalty term:

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

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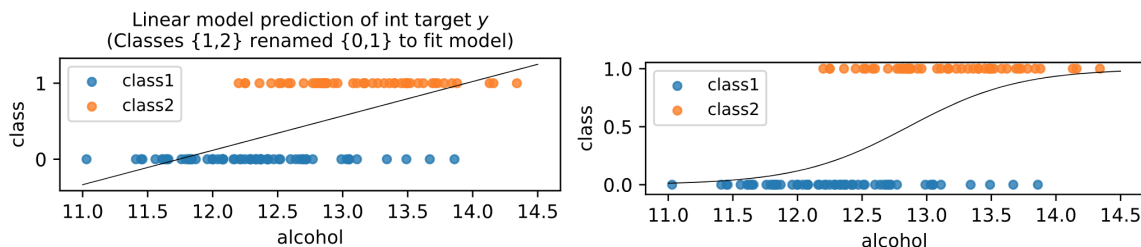
To make a classifier out of such a model, we can simply threshold the probability so that the classifier predicts class 0 when the probability is < 0.5 and predicts class 1 when the probability is ≥ 0.5.

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

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

It is technically still a regressor not a classifier; it only becomes a classifier when we apply a threshold to the resulting probability.
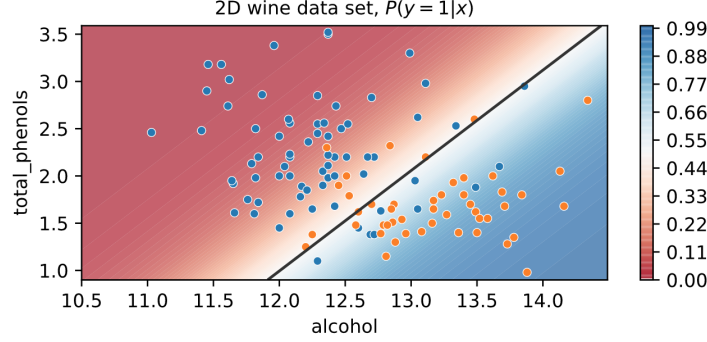
If we apply the sigmoid function to the output of a linear model, we get:

---

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

$$P(y = 1 \,|\, \mathbf{x}') = \sigma(\mathbf{x}'\beta) = \frac{1}{1 + e^{-\mathbf{x}'\beta}}$$

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

Here is what the sigmoid looks like in 2D where the color indicates the $P(y = 1 \,|\, \mathbf{x}')$:

2D wine data set, $P(y = 1|x)$

B.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.

B.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:

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

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 since ESLII drops the negation because they're setting equal to zero.) Pass in the negative of the log likelihood as the loss function and this log likelihood gradient as the loss gradient function. Just to be clear, we use the negative of the loss function so the minimizer maximizes the max likelihood.

B.3   L1 Lasso regularization

As a resource, see a student's implementation of 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 would need to *maximize* (again dropping the $\frac{1}{n}$ as unnecessary for optimization):

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

But, we want to minimize so flip the signs:

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

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's partial derivative differently than for the other p coefficients. To make matters worse, the computation for $\beta_1 .. \beta_p$ involves β_0 and vice versa. ESLII p125 Equation 4.31 simply says to find the β_0 and β that maximize the penalized logistic regression equation. To solve this in practice, we compute β_0's partial derivative separately from those for $\beta_{p \geq 1}$. For β_0, the partial derivative is:

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

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 $\frac{1}{n}$ as it's not necessary for optimization purposes. 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:

$$\frac{\partial}{\partial \beta_0} \mathscr{L}(\beta, \lambda) = \frac{1}{n} x_0 (\mathbf{y} - \sigma(\mathbf{X}' \cdot \beta))$$

but vector variable x_0 is just $\vec{\mathbf{1}}$:

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

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:

$$\frac{\partial}{\partial \beta_0} \mathscr{L}(\beta, \lambda) = mean(\mathbf{y} - \sigma(\mathbf{X}' \cdot \beta))$$

For $\beta_1 .. \beta_p$, we need to carefully identify which parts of $\mathbf{X}'$ and β we are using. To get the error term, $\mathbf{y} - \sigma(\mathbf{X}' \cdot \beta)$, we use the full augmented-with-ones matrix $\mathbf{X}'$ and β. For the other parts of the gradient equation, we are computing coefficients for $\beta_{1..p}$ so we're not using the augmented matrix:

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

16

(Only the inner $\mathbf{X}'$ has the augmented-with-one rows.)

Then the β_0 gradient and these gradients are combined to get a full ∇_β. The algorithm to compute the full gradient is:

Algorithm: *L1NegLogLikelihood($\mathbf{X}', \mathbf{y}, \beta'$)*

$err = \mathbf{y} - \sigma(\mathbf{X}' \cdot \beta')$ (*error vector* is $n \times 1$ *column vector*)

$\frac{\partial}{\partial \beta_0} = mean(err)$ (*usual log-likelihood gradient; use current β'*)

$r = \lambda \, sign(\beta')$ (*regularization term $p + 1 \times 1$ column vector*)

$r[0] = 0$ (*kill β_0 position but keep as $p + 1 \times 1$ vector*)

$\nabla = \frac{1}{n} \left\{ \mathbf{X}'^{T} err - r \right\}$

return $-\begin{bmatrix} \frac{\partial}{\partial \beta_0} \\ \nabla_1 \\ \vdots \\ \nabla_p \end{bmatrix}$

C   Gradient-descent algorithm

The core minimization algorithm function is:

Algorithm: *basic_minimize*($\mathbf{X}$, $\mathbf{y}$, $\mathscr{L}$, $\nabla\mathscr{L}$, η) **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

  $\vec{b} = \vec{b} - \eta\nabla\mathscr{L}$

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*($\mathbf{X}$, $\mathbf{y}$, $\mathscr{L}$, $\nabla\mathscr{L}$, η, γ) **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

  $\vec{v} = \gamma\vec{v} + \eta\nabla\mathscr{L}$ (*Add a bit of previous direction to next direction*)

  $\vec{b} = \vec{b} - \vec{v}$

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 $\mathbf{X}$ 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 L1/L2 Ridge regression because the solution relies on mean-centered x_i: for L1/L2, we do NOT add the $\vec{1}$ vector to augment $\mathbf{X}'$. Your project has an `addB0` argument (but we're only implementing L2 regression so it's the only use of `addB0=False`).

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, which is useful for all variations of linear and logistic regression:

Algorithm: $adagrad\_minimize(\mathbf{X}, \mathbf{y}, \mathscr{L}, \nabla\mathscr{L}, \eta, \epsilon=1e\text{-}5)$ **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
   $\vec{h} \mathrel{+}= \nabla\mathscr{L} \otimes \nabla\mathscr{L}$ ($track$ sum of $squared$ $partials$, use $element$-$wise$ $product$)
   $\vec{b} = \vec{b} - \eta * \frac{\nabla\mathscr{L}}{(\sqrt{\vec{h}}+\epsilon)}$
until $|(\mathscr{L}(\vec{b}) - \mathscr{L}(\vec{b}_{prev}))| < precision$;
return $\vec{b}$

In your implementation, you'll need a parameter, `addB0`, that dictates whether or not to use augmented vectors and matrices. We always set `addB0=True` except for Ridge linear regression, which computes β_0 outside of the minimization process (as just $\overline{y}$).