Gradient Descent

We will be implementing logistic regression using the gradient descent algorithm.

References:

- Andrew Ng's Machine Learning Lecture Notes

Notes on Implementing Gradient Descent:

- Implementing gradient descent can lead to challenging debugging. Try computing values by hand for a really simple example (1 feature, 2 data points) and make sure that your methods are getting the same values.
- Numpy is your friend. Use the power of it! There should only be one loop in your code (in `run`). You should never have to loop over a numpy array. See the numpy tutorial anddocumentation .
- Download data and starter code here.

Part 1: Create Data

1. Generate a dataset using sklearn's make_classification module.

```
X, y = make_classification(n_samples=100,
                           n_features=2,
                           n_informative=2,
                           n_redundant=0,
                           n_classes=2,
                           random_state=0)
```

8. We use two features so that we can visualize our data. Make a scatterplot with the first feature on the x axis and the second on the y axis. Differentiate the positive results from the negative results somehow (different colors or different symbols).
9. Just by eye-balling, make an estimate for a good coefficient for this equation: `y = mx` of the decision boundary. We are using this form

since our first implementation will not find an intercept. But also make an estimate for good coefficients for this equation: y = mx+b.

Part 2: Cost function

In order to be able to evaluate if our gradient descent algorithm is working correctly, we will need to be able to calculate the cost.

Recall the log likelihood function for logistic regression. Our goal is to maximize this value.

$$\ell(\boldsymbol{\beta}) = \sum_{i=1}^{n} y_i \log(h(\mathbf{x_i})) + (1 - y_i) \log(1 - h(\mathbf{x_i}))$$

Recall that the hypothesis function h is defined as follows:

$$h(\mathbf{x_i}) = \frac{1}{1 + e^{-\boldsymbol{\beta}\mathbf{x_i}}}$$

Since we will be implementing Gradient Descent, which minimizes a function, we'll look a the cost function below, which is just the negation of the log likelihood function above.

$$J(\boldsymbol{\beta}) = -\sum_{i=1}^{n} y_i \log(h(\mathbf{x_i})) + (1 - y_i) \log(1 - h(\mathbf{x_i}))$$

The gradient of the cost function is as follows:

$$\nabla J(\boldsymbol{\beta}) = \left[\frac{\partial}{\partial \beta_1} J(\boldsymbol{\beta}), \frac{\partial}{\partial \beta_2} J(\boldsymbol{\beta}), \ldots, \frac{\partial}{\partial \beta_p} J(\boldsymbol{\beta}) \right]$$

Each partial derivative will be computed as follows:

$$\frac{\partial}{\partial \beta_j} J(\boldsymbol{\beta}) = \sum_{i=1}^{n} (h(\mathbf{x_i}) - y_i) \, x_{ij}$$

1. To verify that your implementations are correct, compute the following by hand. Of course, you can use a calculator/google/wolfram alpha/python.

	feature 1	feature 2	y
x_1	0	1	1
x_2	2	2	0
$x_{3/}$	3	0	0

1. Using the data above, compute the value of the cost function. Initialize your coefficients: $\beta_1 = 1$, $\beta_2 = 1$.

   Hint: you will use $(\beta_1 x_{1,1} + \beta_2 x_{1,2})$ while computing your hypothesis function for the first data point.

2. Using the data above, compute the gradient of the cost function.

2. In `logisticregression_functions.py`, implement `predict_proba` and `predict`functions. `predict_proba` will calculate the result of the hypothesis function `h(x)` for the given coefficients. This returns float values between 0 and 1. `predict` will round these values so that you get a prediction of either 0 or 1. You can assume that the threshold we're using is 0.5.

3. In `logisticregression_functions.py`, implement `cost` and `gradient`. You should be able to use the `predict_proba` function you implemented above. Make sure to check that you get the same values as you computed above.

   In a terminal, you should be able to run your function like this:

```
import logisticregression_functions as f
import numpy as np
X = np.array([[0, 1], [2, 2]])
y = np.array([1, 0])
coeffs = np.array([1, 1])
f.cost(X, y, coeffs)
```

   Make sure to do `reload(f)` if you make changes to your code.

Part 3: Implement Gradient Descent

Now are are going to implement gradient descent, an algorithm for solving optimization problems.

Below is pseudocode for the gradient descent algorithm. This is a generic algorithm that can solve a plethora of optimization problems. In our case, β we are solving for is the coefficient vector. We will initialize it to be all zeros.

In this pseudocode and in our implementation, we will stop after a given number of iterations. Another valid approach is to stop once the incremental improvement in the optimization function is sufficiently small.

```
Gradient Descent:
  input: J: cost function
       α : learning rate
       k: number of iterations
  output: local minimum of J

  initialize β  (often as all 0's)
  repeat for k iterations:
     β  <- β  - α  * gradient(J)
```

You are going to be completing the code stub in `GradientDescent.py`.

1. Start by taking a look at the starter code. Note how the `GradientDescent` object is initialized. It takes a cost function and a gradient function. We will pass it the functions that we wrote above. Here's example code of how we'll be able to run the Gradient Descent code once you've completed all the functions.

```
2. import logisticregression_functions as f
3. from GradientDescent import GradientDescent
4.
5. gd = GradientDescent(f.cost, f.gradient, f.predict)
6. gd.run(X, y)
7. print "coeffs:", gd.coeffs
8. predictions = gd.predict(X)
```

9. Implement the `run` method. Follow the pseudocode from above.

10. Implement the `predict` method. It should just call the `predict_func` function that was taken as a parameter.

    This will be a kind of boring function. But later it will get more interesting if we do any preprocessing on the data.

Part 4: Run gradient descent & compare

Now we're ready to try out our gradient descent algorithm on some real data.

1. Run your version of gradient descent on the fake data you created at the beginning.

   Note: If you're having trouble getting it to converge, run it for just a few iterations and print out the cost at each iteration. The value should be going down. If it isn't, you might need to decrease your learning rate. And of course check your implementation to make sure it's correct. You can also try printing out the cost every 100 iterations if you want to run it longer and not get an insane amount of printing.
2. Do you get coefficient values similar to your prediction?
3. Run sklearn's `LogisticRegression` on the fake data and see if you get the same results.

Part 5: Add Intercept

Ideally we would like to also have an intercept. In the one feature case, our equation should look like this: `y = mx + b` (not just `y = mx`). We solve this by adding a column of ones to our feature matrix.

1. Implement `add_intercept` in `logisticregression_functions.py` and use it to modify your feature matrix before running gradient descent.

```
2. def add_intercept(X):
3.     '''
4.     INPUT: 2 dimensional numpy array
5.     OUTPUT: 2 dimensional numpy array
6.
```

```
    Return a new 2d array with a column of ones added as the first
    column of X.
    '''
```

10. Modify the `__init__` method of `GradientDescent` so that it can take a boolean parameter `fit_intercept`:

```
def __init__(self, cost, gradient, fit_intercept=True):
    # code goes here
```

    If you set `fit_intercept` to be False, it should work the same way as before this modification.

13. Check that you get a similar result to your prediction.

Part 6: Scaling

If you try running your gradient descent code on some of the data from yesterday, you'll probably have issues with it converging. You can try playing around with alpha, the learning rate (one option is to decrease the learning rate at every iteration).

An easier way is to scale the data. Basically, we shift the data so that the mean is 0 and the standard deviation is 1. To do this, we compute the mean and standard deviation for each feature in the data set and then update the feature matrix by subtracting each value by the mean and then dividing by the standard deviation.

1. Commit your code! Run a `git commit -m "some message"` so that if you goof things up with your changes you don't lose your previous version.
2. Add the some methods to the `GradientDescent` class to calculate the scale factors and to scale the features (if the parameter is set.
    - Note: Make sure to scale before you add the intercept column. You don't want to try and scale a column of all ones.
3. Modify the `__init__` method of the `GradientDescent` class so that it can take a boolean `scale` parameter. Add calls of the above functions to the `run` method if `scale` is `True`.

4. Make sure to scale the features before you call the `cost` or `gradient` function.
    - Note: You should calculate mu and sigma from the training data and use those values of mu and sigma to scale your test data (that would result in dividing by 0).
5. Run your code on the fake data and make sure you get the same results.
6. Try running your code on the data from yesterday. Does it converge?

Part 7: Regularization

Recall that regularization helps us deal with overfitting. Let's implement L2 regularization (Ridge).

We will be adding the following term to the cost function.

$$\lambda \sum_{j=1}^{p} \beta_j^2$$

1. Again, don't forget to commit first!
2. Modify your cost to include the above term. You should add an additional parameter to the cost function called `lam` which is the lambda in the regularization term.
3. Modify your gradient function to include the gradient of the above term. You should add the `lam` term here as well.
4. When you instantiate the `GradientDescent` object, you will need to use a new version of these two functions. Here's how you can create them:

```
def cost_regularized(X, y, coeffs):
    return cost(X, y, coeffs, lam=1)

def gradient_regularized(X, y, coeffs):
    return gradient(X, y, coeffs, lam=1)
```

> Note that you won't actually need to modify your `GradientDescent` algorithm at all.

Always commit before you start a new part! If you muck up your previously working solution, you'll want to get back to it!

Part 8: Termination

We can instead of terminating after a fixed number of iterations, we can terminate when the incremental improvement in the cost function is sufficiently small.

1. Add a parameter `step_size` to the `run` function. Terminate the loop once the incremental decrease in the cost function is smaller than `step_size`. Note that this means you'll have to evaluate the cost function at each iteration and compare it to the previous value of the cost function. Specify your function to rely on the step_size parameter instead of the num_iterations parameter if the step_size is not None.
2. Figure out what a good value for the `step_size`. If it's too large, you won't make it to the optimal solution. If it's too small, it will take too long to converge.

Part 9: Stochastic Gradient Descent

Stochastic Gradient Descent is a variant of the gradient descent algorithm that in practice converges faster. The difference is that at each iteration, stochastic gradient descent only uses one training example for its update. Here is the pseudocode. `n` here is the number of training examples.

```
Randomly shuffle examples in the training set.
Repeat until step size is sufficiently small:
  for i = 1, 2, ... n:
    β <- β - α * gradient for x_i
```

Here's the cost function for a single data point x_isub>:

$$\frac{\partial J_i(\boldsymbol{\beta})}{\partial \beta_j} = \log(h(\mathbf{x_i})) + (1 - y_i) \log(1 - h(\mathbf{x_i})$$

Here is the formula for the partial derivatives that make up the gradient:

$$\frac{\partial J_i(\boldsymbol{\beta})}{\partial \beta_j} = (h(\mathbf{x_i}) - y_i) \, x_{ij}$$

Note that we shuffle the training examples and then go through them in order so that we use every training example before repeating.

1. Implement stochastic gradient descent. Make sure to save the old version of gradient descent so that we can compare them.
2. Use ipython's timeit to compare the runtime of the standard gradient descent algorithm and stochastic gradient descent. Also compare their results to verify that their performance is the same.

Part 10: Newton's Method for a single variable

While gradient descent and stochastic gradient descent are two of the most common optimization techniques, they are not the only techniques used by data scientists. Newton's Method is a root-finding algorithm. When we apply Newton's Method to the derivative of a function, we can find the roots of the derivative, which is equivalent to finding the extrema of a function.

1. Newton's Method can be used for finding the roots of a differentiable function. You may have learned this algorithm in your high school calculus class. Here is an overview of how the algorithm works.

2. 1. Provide an initial guess x0 for the location of the root.
3.
4. 2. Calculate f(x0)
5.
6. 3. If f(x0) is less than your pre-defined tolerance, then stop. Otherwise continue.
7.
8. 4. Find the x-intercept of the line tangent to f(x) at x0.
9.
10. 5. Update x0 := the x-intercept found in step 4.

```
11.
12. 6. Return to step 2.
```

Here is an [animation of how it works]

Equivalently, this iterative method uses this update formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

13. Newton's Method can also be used to find **local optima of any twice-differentiable function**, such as the cost function, by approximating the zeros of a function's derivative. This technique is also known as Newton's Method in optimization.

    First, we can set

    $$g(x) = f'(x)$$

    Then, our new update formula will take the form

    $$x_{n+1} = x_n - \frac{g(x_n)}{g'(x_n)} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

    And, we continue updating until the derivative, g(x), is less than our tolerance level.

    Note: if our function is quadratic, this method will find the vertex in one step.

14. **Exercise:** Using Newton's Method for optimization, write a function that returns one of the local minimas of the quartic function f(x) = $x^4 + 2x^3 - 5x^2 - 8$.

```
15. def newton_quartic(x0, epsilon):
16. '''
17.   INPUT: initial guess for x0, tolerance level
18.   OUTPUT: cartesian coordinates of minima, number of iterations required
19. '''
```