Week 4 Lab

The questions below are due on Tuesday October 10, 2017; 11:00:00 PM.

We will start by running code for solving regression problems and doing gradient descent on the hinge loss, to develop an understanding for how it behaves. By the end of the lab you will have implemented all of the important functions necessary to implement the demos at the beginning.

1) EXPLORING GRADIENT DESCENT FOR REGRESSION AND CLASSIFICATION

In many problems, we want to predict a real value, such as the actual gas mileage of a car, or the concentration of some chemical. Luckily, we can use most of a mechanism we have already spent building up, and make predictors of the form:

$$y = \theta^T x + \theta_0$$

This is called a *linear regression* model.

We would like to learn a linear regression model from examples. Assume X is a d by n array (as before) but that Y is a 1 by n array of floating-point numbers (rather than +1 or -1). Given data (X, Y) we need to find θ, θ_0 that does a good job of making predictions on new data drawn from the same source.

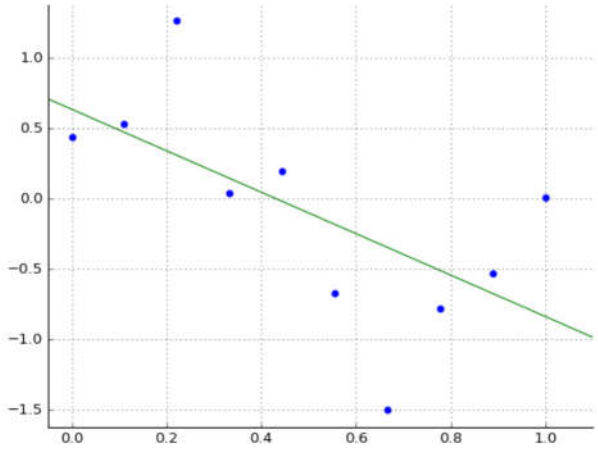
We will approach this problem by formulating an objective function. There are many possible reasonable objective functions that implicitly make slightly different assumptions about the data, but they all typically have the form:

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} L(x^{(i)}, y^{(i)}, \theta, \theta_0) + \lambda R(\theta, \theta_0)$$

For regression, we most frequently use *squared loss*, in which

$$L_s(x, y, \theta, \theta_0) = (y - \theta^T x - \theta_0)^2$$

In this lab, we will experiment with linear regression, gradient descent, and polynomial features. We will start with a simple data-set with one input feature and 10 training points, which is easy to visualize.



1.1) Least Squares Regression

Recall from the lecture and homework that it is possible to solve a least-squares regression problem directly via the matrix algebra expression for the parameter vector θ in terms of the input data X and desired output vector Y. In this section, we will explore this *analytic* solution strategy. For the purposes of this problem, we will assume that the feature repreesentation of the data includes a constant feature with the value 1, and therefore not include an additional offset parameter θ_0.

The function `t1`, by default, computes and displays the analytic solution to the least squares regresion problem, as shown:

```
1 def run():
2     return t1(order=1)
3
```

Ask for Help

We can try to get a better fit to the data by moving to a higher-dimensional feature space using the polynomial basis. In this example, since there is only a single input feature, which we will call x, it will be transformed into the vector $(x^0, \ldots, x^k)$ if we are using a k-th order basis. (Remember that x^0 is 1!)

1. What is the dimension of a 1st-order polynomial basis?
2. What is another way to explain the θ value that results when using the 0th order basis?

Now, experiment with the function `t1` (in the question box above), which takes a single positive integer argument, which is the order of the polynomial basis to use as the feature transformation.

3. Polynomial order 9 is particularly interesting. Explain what is happening there.
4. How does the magnitude of θ change with order? (You don't have to answer this quantitatively)
5. What polynomial order do you feel represents the hypothesis that will be the most predictive?
6. When we run 10-fold cross validation on this data set, varying the order from 0 to 10, we get the following mean squared error results:

```
[0.69206670716618535, 0.53820006043438084, 0.73424762793041687, 0.283549557896119
3, 0.74338564580774558, 0.61422551802155112, 6.156711267187811, 356.8873742619765
, 408.34678081302491, 786.77402624476542, 1597.6416924085727]
```

What is the best order, based on this data? Does it agree with your previous answer?

1.2) Regularizing the parameter vector

If we add a squared-norm regularizer to the empirical risk, we get the so-called *ridge regression* objective:

$$J_{ridge}(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^{n} L_s(x^{(i)}, y^{(i)}, \theta, \theta_0) + \lambda\|\theta\|^2$$

It's a bit tricky to solve this analytically, because you can see that the penalty is on θ but not on θ_0, so we will not do the derivation. (We generally don't penalize θ_0, so that the overall mean regression value is not penalized. You might be concerned about the fact that there is also an element in θ associated with a constant feature 1, which is effectively an offset and is being penalized. That is true, but not really a problem, since θ_0 is unregularized and can take on any value that is effedtive.)

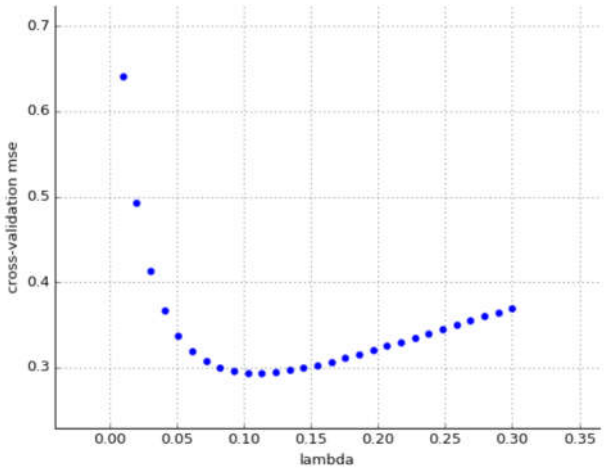
The procedure `t2(order, lam)` performs ridge regression on polynomial features of order `order` with regularization coefficient `lam`. In orange, it draws the original least-squares solution, and in green, it draws the regularized version.

```
1 def run():
2     return t2(order=9, lam = 1)
3
```

Ask for Help

Using a 9-th order polynomial feature function, experiment with λ.

7. What happens with very large (e.g., infinite) and very small (0) values of λ?
8. What value of λ do you feel will give good performance on new data from the same source?
9. When we run 10-fold cross validation on this data set, using 9-th order features, varying λ from 0.01 to 0.3, we get the following plot:



What is the best value of λ, in terms of generalization performance, based on this data? Does it agree with your previous answer? Is it the same as the best value for performance on the training set?

1.3) Gradient descent

Computing the analytic solution requires inverting a d by d matrix; as the size of the feature space increases, this becomes difficult. In addition, there are lots of other useful machine-learning models for which there is no closed-form analytic solution. So, we'll play with gradient descent here and see how it works. The procedure

```
t3(order, lam, step_size, max_iter)
```

performs gradient descent on the ridge regression objective using polynomial features of order `order`. You can specify the maximum number of iterations(we capped it at 10,000 to keep from killing the server) and the step size (we found that a fixed step size tended to work better than an a simple decreaeing one). It will print a convergence plot (objective value versus iteration number) and then show the solution it found in green and the analytic solution in orange for comparison.

```
1 def run():
2     return t3(order=2, lam=0, step_size= 0.3, max_iter = 1000)
3
```

Ask for Help

Keeping $\lambda = 0$, experiment with this method for solving regression problems.

10. What step sizes were needed to match the analytic solution almost exactly for 1st and 2nd order bases?
11. What step size allowed you to get the same shape as the analytical solution for 3rd order? (Use the convergence plot to help decide on step size and max iterations.)
12. For 9th order, play around for a while (no more than 10 minutes), to get a result as close to the analytic solution as you can. How does it compare to the analytical solution?

1.4) Pegasos

Last week we experimented with gradient descent applied to the SVM objective function. Currently, for each gradient-descent step, we sum the loss over all the data points. If we have a really big data set (e.g., the reviews from lab 2) then even doing one update will be very expensive.

Let's do a small re-write of the SVM objective:

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \left(L_h(y^{(i)}(\theta^T x^{(i)} + \theta_0)) + \lambda \|\theta\|^2 \right)$$

Note that the regularization term is now inside the sum, so it's getting added in n times; but it's also captured by the $1/n$ on the outside of the sum, so we're okay. This means that if we were to just sum up the gradient of the quantity inside the sum, and divide by n, we'd get the same update as before.

Now, we're going to consider what seems like a slightly crazy thing. Instead of summing over all the training examples to do an update, we're going to pick one at random from our input data, do an update, pick another one, do an update, etc. This is called *stochastic* gradient descent. We don't need to worry about the $1/n$ because it can get absorbed into the step size (but we will generally need a much smaller step size and a bigger T to get this to work). Theoretically, if you decay the step size appropriately, it is still guaranteed to converge to a local optimum of the original objective. And, it has some other desirable and undesirable properties:

- Each step is computationally efficient
- The update based on a data point is done based on the effects of the points we have processed so far
- The updates are kind of "noisy" because the solution is moved in different directions by different data points
- In aggregate, the stochastic updates move the parameters in (nearly) the same direction as the gradient of the full objective
- The noisiness of the update sometimes helps get us out of shallow local minima

In many cases, people use an intermediate procedure, called a "mini-batch" update, which draws k training examples at random and then does an update that sums the gradient over those k points.

The Pegasos algorithm described in the Week 3 notes (Algorithm 1, page 8) is basically stochastic gradient descent on the SVM objective, with adaptive step size. This is a very common pattern that we will see again, for example, when we study neural network training.

We considered several implementations of linear- classifier learning. Pegasos is an implementation of the algorithm as written in the notes. GD-100 applies batch gradient descent (our `gd` code with max_iter=100) on the SVM objective. SGD applies stochastic gradient descent (our `sgd` code, with max_iter=5*n (n is number of points)) on the SVM objective. Some care went into making sure that the function that computes the value and the gradient (both for `gd` and `sgd`) was reasonably efficient, with no duplicated computations. Finally, by way of contrast, we compared against `perceptron` and `averaged_perceptron` (this is the simple version of averaged perceptron from Lab 1, not the optimized one from the notes).

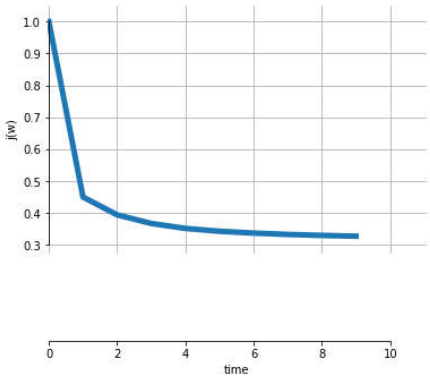
Below are the accuracies from 5-fold cross-validation on the data sets in lab 2. Auto is the MPG data from lab 2, with a reasonable choice of feature representation. Rev1 is a small subset of the review data (1000 reviews) and Rev2 is the full (10,000 reviews) dataset. The blank entry indicates the the algorithm took way too long to run...

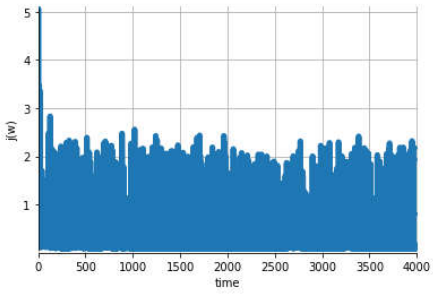
Algorithm	Auto Accuracy	Rev1 Accuracy	Rev2 Accuracy
Pegasos	0.908	0.750	0.825
GD-100	0.905	0.752	--
SGD	0.918	0.750	0.823
Avg Perc	0.916	0.759	0.826
Perc	0.898	0.748	0.792

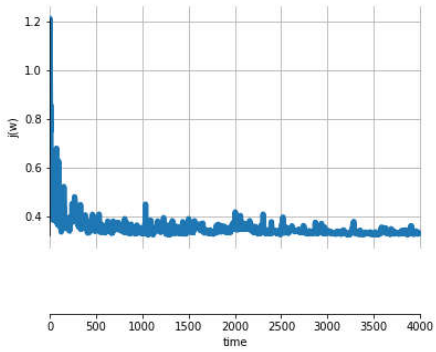
Below are the running times, in seconds, of the various algorithms.

Algorithm	Auto Time	Rev1 Time	Rev2 Time
Pegasos	0.080	4.320	21.720
GD-100	0.062	68.475	--
SGD	0.232	4.320	59.180
Avg Perc	0.052	1.223	15.187
Perc	0.050	0.962	12.742

The following plots show the convergence of the gradient descent implementations on the auto MPG data. The leftmost plot shows the value of the SVM objective for the first 10 iterations of gradient descent. The middle plot shows the value of the SVM objective for the randomly sampled point in each iteration of stochastic gradient descent. The rightmost plot shows the SVM objective over the whole data set for the θ parameters at each iteration of the same sgd run.







13. Discuss the relative accuracies of the algorithms.
14. Explain the relative performance of the algorithms.
15. Explain the convergence plots.

1.5) Checkoff

Checkoff 1:

Have a check-off conversation with a staff member, to explain your answers.

Ask for Help | Ask for Checkoff

2) LINEAR REGRESSION - GOING DOWNHILL

1) We will now write some general Python code to compute the gradient of the squared-loss objective, following the structure of the expression and the rules of calculus. Note that this style of writing the gradient functions maps directly into the chain-rule steps required to compute the gradient, but produces code that is inefficient, because of duplicated computations. It is straightforward to implement more efficient versions if you want to use them for larger problems.

We start by defining some basic functions for computing the mean squared loss. Note that we want these to work for any value of n, that is, `x` could be a single feature vector or a full data matrix and similarly for `y`.

```
# In all the following definitions:
# x is d by n : input data
# y is 1 by n : output regression values
# th is d by 1 : weights
# th0 is 1 by 1 or scalar
def lin_reg(x, th, th0):
    return np.dot(th.T, x) + th0
def square_loss(x, y, th, th0):
    return (y - lin_reg(x, th, th0))**2
def mean_square_loss(x, y, th, th0):
    # the axis=1 and keepdims=True are important when x is a full matrix
    return np.mean(square_loss(x, y, th, th0), axis = 1, keepdims = True)
```

These functions will already be defined when you are answering the questions below.

Warm up:

If X is d by n and Y is 1 by n, what is the dimension of θ? d by 1

Ask for Help

If X is d by n and Y is 1 by n, what is the dimension of $\nabla_\theta J_{emp}(\theta, \theta_0)$? d by 1

Ask for Help

2) Now let's compute the gradients with respect to θ, make sure that they work for data matrices and label vectors. You can write one function at a time, some of the checks will apply to each function independently.

```
# Write a function that returns the gradient of lin_reg(x, th, th
# with respect to th
def d_lin_reg_th(x, th, th0):
    return None

# Write a function that returns the gradient of square_loss(x, y,
# respect to th.  It should be a one-line expression that uses li
# d_lin_reg_th.
def d_square_loss_th(x, y, th, th0):
    return None

# Write a function that returns the gradient of mean_square_loss(
# respect to th.  It should be a one-line expression that uses d_
def d_mean_square_loss_th(x, y, th, th0):
    return None

```

Ask for Help

3) Now let's compute the gradients with respect to θ_0, make sure that they work for data matrices and label vectors. You can write one function at a time, some of the checks will apply to each function independently.

```
# Write a function that returns the gradient of lin_reg(x, th, th
# with respect to th0
def d_lin_reg_th0(x, th, th0):
    return None

# Write a function that returns the gradient of square_loss(x, y,
# respect to th0.  It should be a one-line expression that uses l
# d_lin_reg_th0.
def d_square_loss_th0(x, y, th, th0):
    return None

# Write a function that returns the gradient of mean_square_loss(
# respect to th0.  It should be a one-line expression that uses d
def d_mean_square_loss_th0(x, y, th, th0):
    return None

```

Ask for Help

3) GOING DOWN THE RIDGE

Now, let's add a regularizer. The ridge objective can be implemented as follows:

```
# In all the following definitions:
# x is d by n : input data
# y is 1 by n : output regression values
# th is d by 1 : weights
# th0 is 1 by 1 or scalar
def ridge_obj(x, y, th, th0, lam):
    return np.mean(square_loss(x, y, th, th0), axis = 1, keepdims = True) + lam * np.linalg.norm(th)**
2
```

Let's extend our previous code for the gradient of the mean square loss to compute the gradient of the ridge objective with respect to θ. Our previous solutions for the non-ridge case: `d_mean_square_loss_th` and `d_mean_square_loss_th0` are defined for you and you can call them.

```
1 def d_ridge_obj_th(x, y, th, th0, lam):
2     return None
3
4 def d_ridge_obj_th0(x, y, th, th0, lam):
5     return None
6
```

Ask for Help

4) STOCHASTIC GRADIENT

We will now implement stochastic gradient descent in a general way, similar to what we did with gradient descent (`gd`).

The calling conventions for `sgd` are similar to those of `gd` except that we need to pass in the data and labels for the problem. Similarly, the objective function for each step needs a point and a label as well as an x.

- `X`: a standard data array (d by n)
- `y`: a standard labels row vector (1 by n)
- `JdJ`: a function whose input is a data point (a column vector), a label (1 by 1) and a weight vector `w` (a column vector), and which returns a tuple `(f, df)`, where `f` is a scalar and `df` is a column vector.
- `w0`: an initial value of w, which is a column vector.
- `step_size_fn`: a function that is given the iteration index (an integer) and returns a step size. *

- `max_iter`: the number of iterations to perform

It returns a tuple (like `gd`):

- `w`: the value at the final step
- `fs`: the list of values of f found during all the iterations
- `ws`: the list of values of w found during all the iterations

You might find the function `np.random.randint(n)` useful in your implementation.

Hint: This is a short function; our implementation is around 15 lines.

The test cases are:

```
import numpy as np

def super_simple_separable():
    X = np.array([[2, 3, 9, 12],
                  [5, 2, 6, 5]])
    y = np.array([[1, -1, 1, -1]])
    return X, y

X, y = super_simple_separable()

def testloss(true_label, predicted, eps=1e-6):
  p = np.clip(predicted, eps, 1 - eps)
  if true_label == 1:
    return -np.log(p)
  else:
    return -np.log(1 - p)

def JdJ(Xi, yi, w):
    def f(w): return testloss(yi, np.dot(w.T, Xi))
    g = num_grad(f)
    return float(f(w)), g(w)
```

```
1 import numpy as np
2
3 def sgd(X, y, JdJ, w0, step_size_fn, max_iter):
4     pass
5
```

Ask for Help

5) PEGASOS: STOCHASTIC GRADIENT DESCENT ON THE SVM OBJECTIVE

The Pegasos algorithm described in the Week 3 notes (Algorithm 1, page 8) is basically stochastic gradient descent on the SVM objective, with adaptive step size. We will consider a slightly generalized version that includes an offset term in the separator.

Given the definitions of `sgd` (above), and `svm_obj` and its gradient (from HW4, Prob 5), we can do a very simple implementation of this generalized Pegasos. We have chosen the arguments and return values to be compatible with our perceptron implementations. In particular, `Pegasos` should return a tuple specifying a linear separator:

- `th`: a dx1 column vector (normal)
- `th0`: a 1x1 column vector (offset)

You might find `np.vstack` useful in your implementation.

```
 8         df = lambda X, Y, w, lam: np.vstack((d_hinge_loss_th(X
 9
10                                             d_hinge_loss_th0(
11                                             + 2*lam*np.linalg
12
13         return f, df
14     return JDJ
15
16
17
18 def Pegasos(data, labels, lam, max_iter):
19     d, n = data.shape
20     JdJ = JdJ_lam(lam)
21     # Do not change these parameters
22     w,_,_ = sgd(data, labels, JdJ, cv((d+1)*[0.]), lambda i: .
23                 max_iter)
24     th, th0 = w[:-1],w[-1]
25     return th, th0
26
```

Ask for Help