PS1-asymloss-notebook

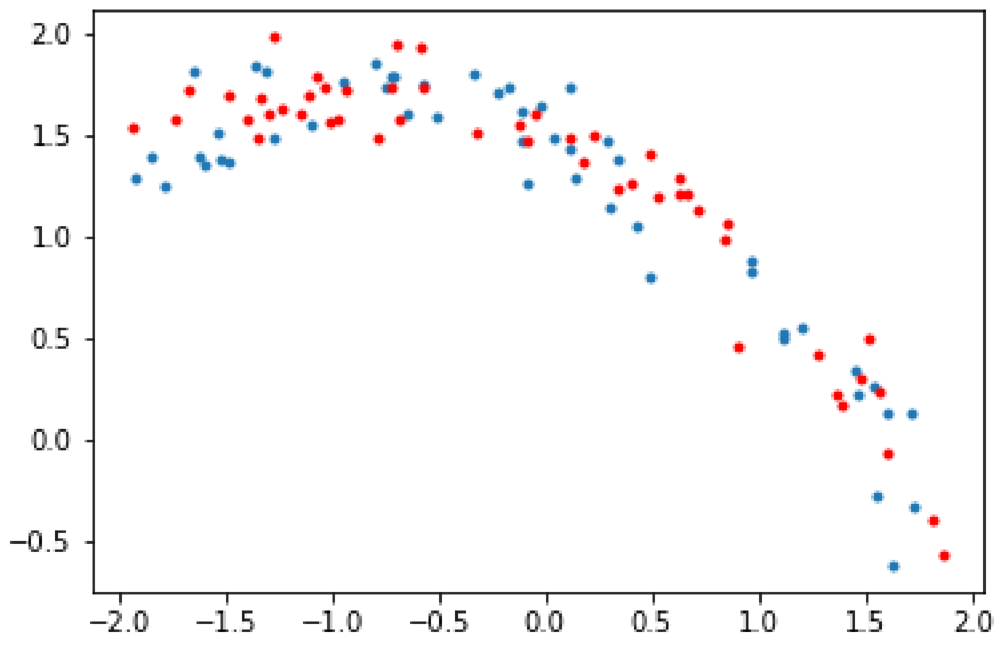
February 27, 2017

1 Problem set 1, Asymmetric loss regression

First, Let's load the data and plot the training data out.

```
In [5]: from __future__ import absolute_import
        from __future__ import division
        from __future__ import print_function
        import numpy as np
        %matplotlib inline
        from utils import *  # this loads definitions of functions etc. in utils.py
        import pylab

        # Load data
        X_train, y_train = loadData('train')
        X_val, y_val = loadData('val')

        pylab.plot(X_train, y_train, '.')
        pylab.plot(X_val, y_val, 'r.')
```

```
Out[5]: [<matplotlib.lines.Line2D at 0x7ecdcc0>]
```



First, we will solve the linear regression (under the “normal” symmetric squared loss) using the closed form solution. Let’s define a couple of functions we will need.

```
In [41]: def symmLoss(X, w ,y):
             """
             Get the symmetric squared loss given data X, weight w and ground trut

             Parameters
             ----------
             X : 2D array
                 N x d+1 data matrix (row per example)
             w : 1D array
                 d+1 length vector
             y : 1D array
                 Observed function values

             Returns
             -------
             loss : a scalar
                 The loss calculated by the symmetric loss formula
             """

             return np.sum([(np.dot(X, w) - y)**2]) / (np.shape(y))

In [23]: def lsqClosedForm(X, y):
             """
             Use closed form solution for least squares minimization
```

```
            Parameters
            ----------
            X : 2D array
                N x d+1 data matrix (row per example)
            y : 1D array
                Observed function values

            Returns
            -------
            w : 1D array
                d+1 length vector
            """
            return np.dot(np.linalg.pinv(X), y)
```

Test the closed form solution: generate a toy data set from a random linear function with no noise. We should be able to perfectly recover w in this case (up to numerical precision).

```
In [24]: X = np.hstack((np.ones([20,1]),np.random.random((20,1))))
         w = np.random.random((2))
         y = np.dot(X,w)
         print('true weight:  '+repr(w))
         w_ = lsqClosedForm(X, y)
         print('function output: '+repr(w_))
         if (np.allclose(w,w_)):
             print('Close enough')
```

```
true weight:  array([ 0.63914461,  0.9399298 ])
function output: array([ 0.63914461,  0.9399298 ])
Close enough
```

The function to estimate the variance of the noise and the log likelihood of the data.

```
In [92]: def logLikelihood(X, w, y):
             """
             Get the estimated variance, and the log likelihood of the data

             Parameters
             ----------
             X : 2D array
                 N x d+1 design matrix (row per example)
             w : 1D array
                 d+1 length vector
             y : 1D array
                 Observed function values

             Returns
             -------
```

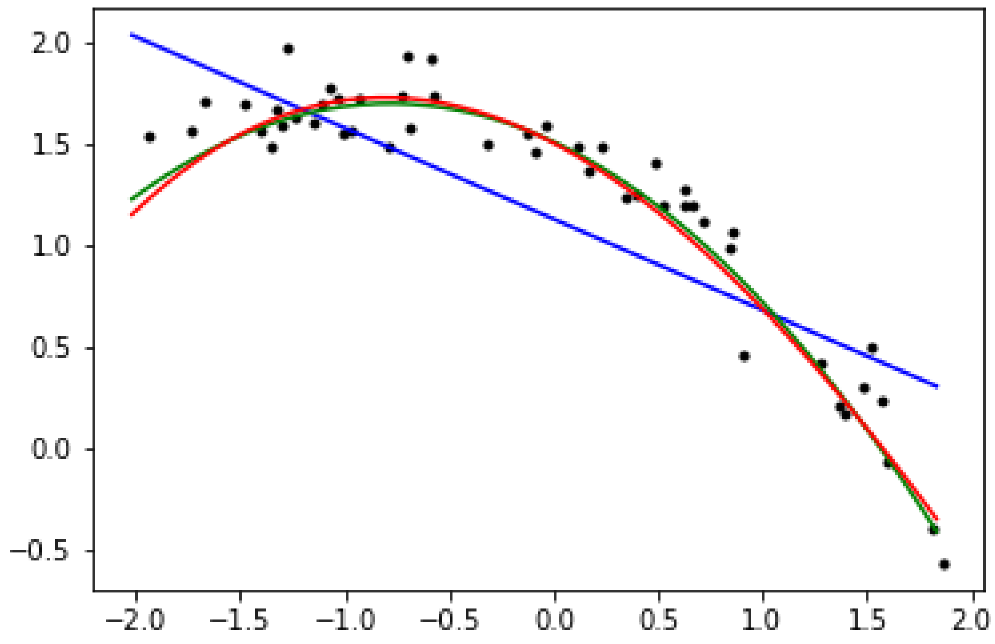
```
            simga2 : a scalar
                The estimated variance (sigma squared)
            loglike : a scalar
                The log-likelihood under the Gaussian noise model N(0,sigma2)
            """
            N = X.shape[0]    # number of rows in X
            # now estimate the variance of the Gaussian noise (sigma2 stands for '
            error = y - np.dot(X,w)
            mean_error = np.mean((y - np.dot(X, w)))
            sigma2 = (1 / float(N-1)) * np.sum((error - mean_error)**2)
            # normalized log-likelihood (mean of per-data point log-likelihood of
            loglike = - 1 / 2.0 * np.log(2*np.pi*sigma2) - np.mean((np.dot(X, w) -
            return sigma2, loglike
```

Now let's fit linear, quadratic and cubic models to the training data, and plot the fit function.

```
In [93]: min_loss = np.Inf
         pylab.plot(X_val, y_val, 'k.')

         # Try degree 1 to 3
         for deg in [1,2,3]:
             # Expand data first; you can check how this function works in utils.py
             X, C = degexpand(X_train, deg)

             # Get the result by applying normal equation
             w = lsqClosedForm(X, y_train)

             # compute loss on training
             loss = symmLoss(X, w, y_train)

             # compute loss on val; note -- use the same scaling matrix C as for tr
             val_loss = symmLoss(degexpand(X_val, deg, C)[0], w, y_val)
             print('degree %d:' %(deg))
             print('train loss %.6f' %(loss))
             print('val loss %.6f' %(val_loss))
             print('sigma^2: %.6f \nlog-likelihood %.6f\n' %logLikelihood(X, w, y_t

             if val_loss < min_loss:
                 min_loss = val_loss
                 # record in best_param the model weights, degree, and the scaling
                 best_param = (w, deg, C)

             # Plot the function
             color = {1:'b', 2:'g', 3:'r'}[deg]
             pylab.plot(np.linspace(min(X_train)-.1,max(X_train)+.1), np.dot(degexp
```

```
degree 1:
train loss 0.155770
```

```
val loss 0.108249
sigma^2: 0.158948
log-likelihood -0.489351

degree 2:
train loss 0.032250
val loss 0.024095
sigma^2: 0.032908
log-likelihood 0.298083

degree 3:
train loss 0.031714
val loss 0.026440
sigma^2: 0.032362
log-likelihood 0.306453
```



```
In [129]: # Announce result on test data
          X_test, y_test = loadData('test')
          print("Best degree:"+repr(best_param[1]))
```

```
Best degree:2
```

```
      -----------------------------------------------------------------------------------
```

```
    TypeError                                 Traceback (most recent call last)

    <ipython-input-129-e91bd5ae8550> in <module>()
      2 X_test, y_test = loadData('test')
      3 print("Best degree:"+repr(best_param[1]))
----> 4 val_loss = symmLoss(degexpand(X_val, best_param[2], best_param[2])[0],


    C:\Users\tmelo1\OneDrive\Documents\Machine Learning\Homework 1\cs475hw1s17-
     61
     62     # Make polynomials
---> 63     out_X = (X[..., np.newaxis] ** (1. + np.arange(deg))).reshape(n, -1
     64
     65     # Add column of ones


    TypeError: only length-1 arrays can be converted to Python scalars
```

Now we want to repeat the experiment above but under the asymmetric loss function. Since there is no closed form solution, we will need to rely on gradient descent. First we need to implement the loss function and the gradient function.

```
In [99]: def asymmLoss(X, w, y,alpha):
             """
             Get the asymmetric loss given data X, weight w and ground truth y

             Parameters
             ----------
             X : 2D array
                 N x d+1 design matrix (row per example)
             w : 1D array
                 d+1 length vector
             y : 1D array
                 Observed function values
             alpha : scalar
                 weight put on positive error, i.e., yhat > y

             Returns
             -------
             loss : a scalar
                 The loss calculated by equation in problem set 1
             """

             # it may be more convenient to define the loss as a per-data point we
             # with weights determined by the sign of the error and collected into
```

```
    N = X.shape[0]
    yhat = np.dot(X, w)
    weights=np.zeros([N,N])
    for i in range(0,N):
        if yhat[i] >= y[i]:
            weights[i][i] = alpha
        else:
            weights[i][i] = 1
    loss = np.sum(np.dot(weights, (y-yhat)**2)) / N
    return loss
```

```
In [112]: def asymmGrad(X, w, y,alpha):
              """
              Get the gradient of w

              Parameters
              ----------
              X : 2D array
                  N x d+1 design matrix (row per example)
              w : 1D array
                  d+1 length vector
              y : 1D array
                  Observed function values
              alpha : scalar
                  weight put on positive error, i.e., yhat > y

              Returns
              -------
              grad : 1D array
                  d+1 length vector
              """
              N = X.shape[0]
              yhat = np.dot(X, w)
              # use the weights here as well, defined by alpha
              weights=np.zeros([N,N])
              for i in range(0,N):
                  if yhat[i] > y[i]:
                      weights[i][i] = alpha
                  else:
                      weights[i][i] = 1
              grad = np.zeros(len(w))
              for i in range(len(w)):
                  grad[i] = -2/N * np.sum(np.dot(weights, (y - yhat)) * X[:,i])
              return grad
```

Test the loss and gradient function. You can manually verify that for the given values of X (2 data points), y and w, with alpha=10, you should get these numbers for the objective (asymmetric loss) value and for the gradient. Then, run the code to make sure your implementation of the gradient is correct.

```
In [113]: loss, grad = asymmLoss(np.array([[1,2], [1, -2]]), np.array([1,1]), np.ar
                      asymmGrad(np.array([[1,2], [1, -2]]), np.array([1,1]), np.arr
          print("expected output")
          print("5.5")
          print("[ 9 -22]")
          print("function output:")
          print(loss)
          print(grad)
```

```
expected output
5.5
[ 9 -22]
function output:
5.5
[  9. -22.]
```

Since we have had the functions to calculate loss and gradient, we can implement the gradient descent algorithm.

```
In [114]: def gradDescent(X, y,alpha,tol=1e-4,maxIt=10000):
              """
              Use gradient descent to min(loss(X, w, y))

              Parameters
              ----------
              X : 2D array
                  N x d+1 design matrix (row per example)
              y : 1D array
                  Observed function values

              Returns
              -------
              w : 1D array
                  d+1 length vector

              it: number of iterations until convergence
              """
              # Random initialize the weight
              w = np.random.randn(X.shape[1])
              lr = 0.01 # learning rate (make it constant 0.01; feel free to experi
              it = 0 # iteration count
              lastloss = np.Inf # loss computed at previous check point
              checkit = 500 # interval to check convergence
              while True:
                  loss, grad = asymmLoss(X, w, y,alpha), asymmGrad(X, w, y,alpha)
```

```
                w = w - (lr * grad)


                it += 1 # advance iteration count

                if it % checkit == 0: # check point -- evaluate progress and deci
                    converged = it >= maxIt or loss > lastloss-tol
                    lastloss = loss
                    print('iter %d:  loss %.4f' %(it,loss))
                    if converged:
                        break

            return w, it
```

Test gradient descent using any alpha and data generated by (random) noiseless linear model; we should recover the true w fairly accurately (although possibly with less accuracy than the closed form solution for alpha=1)

```
In [119]: X = np.hstack((np.ones([20,1]),np.random.random((20,1))))
          w = np.random.random((2))
          y = np.dot(X,w)
          print('true weight:'+repr(w))
          w_, it_ = gradDescent(X, y,10,1e-6,10000)
          print('%d iterations' %it_)
          print('function output:'+repr(w_))
```

```
true weight:array([ 0.17153394,  0.3267031 ])
iter 500:  loss 0.0011
iter 1000:  loss 0.0000
iter 1500:  loss 0.0000
iter 2000:  loss 0.0000
2000 iterations
function output:array([ 0.17159594,  0.32643357])
```

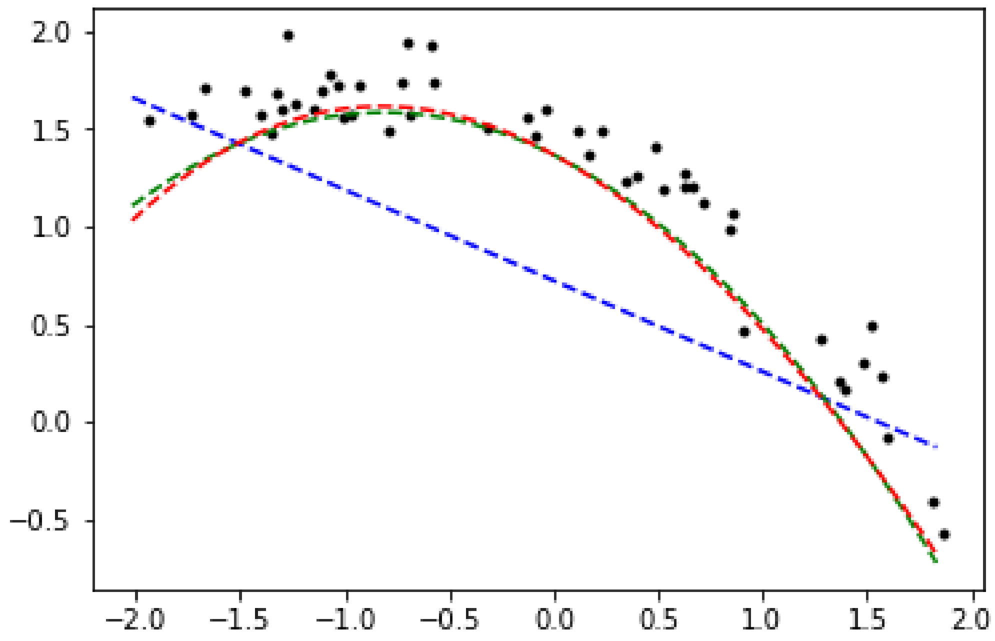
Now we can fit different models, and evaluate their performance on train and val

```
In [120]: min_loss = np.Inf
          # Try degree 1 to 3
          pylab.plot(X_val, y_val, 'k.')
          for deg in [1,2,3]:
              # Expand data first; you can check how this function works in utils.p
              X, C = degexpand(X_train, deg)
              y = y_train

              # Do gradient descent
              w, _ = gradDescent(X, y,10)
              loss = asymmLoss(X, w, y,10)
```

```
            val_loss = asymmLoss(degexpand(X_val, deg, C)[0], w, y_val,10)

            print('degree %d:' %(deg))
            print('train loss %.6f' %(loss))
            print('val loss %.6f' %(val_loss))
            print('sigma^2: %.6f \nlog-likelihood %.6f\n' %logLikelihood(X, w, y_



            if val_loss < min_loss:
                min_loss = val_loss
                best_param = (w, deg, C)

            # Plot the function
            color = {1:'b', 2:'g', 3:'r'}[deg]
            pylab.plot(np.linspace(min(X_train)-.1,max(X_train)+.1), np.dot(degex
```

```
iter 500:  loss 0.4564
iter 1000:  loss 0.4564
degree 1:
train loss 0.456437
val loss 0.374340
sigma^2: 0.159223
log-likelihood -1.005706

iter 500:  loss 0.1550
iter 1000:  loss 0.0900
iter 1500:  loss 0.0890
iter 2000:  loss 0.0890
degree 2:
train loss 0.089027
val loss 0.076941
sigma^2: 0.036509
log-likelihood -0.122804

iter 500:  loss 0.1032
iter 1000:  loss 0.0941
iter 1500:  loss 0.0917
iter 2000:  loss 0.0903
iter 2500:  loss 0.0895
iter 3000:  loss 0.0890
iter 3500:  loss 0.0887
iter 4000:  loss 0.0886
iter 4500:  loss 0.0885
degree 3:
train loss 0.088523
val loss 0.078923
```

```
sigma^2: 0.036320
log-likelihood -0.114172
```



```
In [128]: # Announce result on test data
          X_test, y_test = loadData('test')
          print("Best degree:"+repr(best_param[1]))
```

```
Best degree:2
0.0789226052647
```

```
In [ ]:
```