CS 475 Machine Learning (Spring 2017): Assignment 1

Due on February 27th, 2017 at 3:00PM

Raman Arora

Instructions: Please read these instructions carefully and follow them precisely. Feel free to ask the instructor if anything is unclear!

1. Please submit your solutions electronically via Gradescope.
2. Please submit a PDF file for the written component of your solution including derivations, explanations, etc. You can create this PDF in any way you want: typeset the solution in LATEX (recommended), type it in Word or a similar program and convert/export to PDF, or even hand write the solution (legibly!) and scan it to PDF. We recommend that you restrict your solutions to the space allocated for each problem; you may need to adjust the white space by tweaking the argument to `\vspace{xpt}` command. Please name this document <firstname-lastname>-sol1.pdf.
3. Submit the empirical component of the solution (Python code and the documentation of the experiments you are asked to run, including figures) in a Jupyter notebook file.
4. **Late submissions:** You have a total of 72 late hours for the entire semester that you may use as you deem fit. After you have used up your quota, there will be a penalty of 50% of your grade on a late homework if submitted within 48 hours of the deadline and a penalty of 100% of your grade on the homework for submissions that are later than 48 hours past the deadline.
5. **What is the required level of detail?** When asked to derive something, please clearly state the assumptions, if any, and strive for balance: justify any non-obvious steps, but try to avoid superfluous explanations. When asked to plot something, please include the figure as well as the code used to plot it (and clearly explain in the README what the relevant files are). If multiple entities appear on a plot, make sure that they are clearly distinguishable (by color or style of lines and markers). When asked to provide a brief explanation or description, try to make your answers concise, but do not omit anything you believe is important. When submitting code, please make sure it's reasonably documented, and describe succinctly in the written component of the solution what is done in each `py`-file.

Name: ______________________________

I. Regression

In this set of problems we will look at the regression problem and the maximum likelihood (ML) approach, with the goal to understand a bit better some of their properties.

We will start with simple linear regression, defined by

$$\hat{y} = f(\mathrm{x}; \mathrm{w}) = \mathrm{w} \cdot \mathrm{x} = \sum_{j=0}^{d} w_j x_j \tag{1}$$

It is assumed in (1) that we have augmented the inputs $\mathrm{x} \in \mathbb{R}^d$ by adding 1 as the "zeroth" dimension, $x_0 \equiv 1$. This assumption is valid for the remainder of this problem set, unless stated otherwise.

Assume that we are given n training examples $(\mathrm{x}_1, y_1), (\mathrm{x}_2, y_2), \ldots, (\mathrm{x}_n, y_n)$, where each input data point x_i has d real-valued features. The goal of regression is to learn to predict y from x. We refer to $\mathrm{X} = [\mathrm{x}_1, \cdots \mathrm{x}_n]^\top \in \mathbb{R}^{n \times d}$ as the design matrix and $\mathrm{y} = [y_1, \cdots, y_n]^\top$ as the response vector.

Let $\widehat{\mathrm{w}}$ be the linear regression parameters estimated using the least squares procedure from data, and w^* be the best possible linear regression parameters for the underlying $p(\mathrm{x}, y)$.

It was shown in the lectures that prediction errors made by $\widehat{\mathrm{w}}$ are uncorrelated with the training $\{\mathrm{x}_i\}$, and a more general statement was made but not proven: that the prediction errors are in fact uncorrelated with values of any linear function of x computed on the training inputs.

Later, we made use of a similar fact regarding the prediction errors made by w_*. Here we will prove this more general fact. First, to make things precise, let us recall the definition of correlation between two (continuous) random variables U and V . Let $p_{u,v}$ be their joint probability density, and p_u and p_v the corresponding marginal densities. Then, the correlation (coefficient) between U and V is defined as

$$\rho(U, V) \equiv \frac{\mathbb{E}_{p_{u,v}}[(U - \mathbb{E}_{p_u}[U])(V - \mathbb{E}_{p_v}[V])]}{\sqrt{\mathrm{var}(U)\ \mathrm{var}(V)}} \tag{2}$$

where $\mathrm{var}(U)$ and $\mathrm{var}(V)$ are the variances of U and V under their respective marginal distributions. Intuitively, correlation measures how well one variable is linearly predictable from the other. We will now prove a fact from which it follows that prediction errors of w_* are uncorrelated with any linear function of x:

Problem 1 [15 points] Show rigorously that for any $a \in \mathbb{R}^{d+1}$,

$$\mathbb{E}_{p(\mathrm{x},y)}\left[(y - \mathrm{w}_*{}^\top \mathrm{x})\mathrm{a}^\top \mathrm{x}\right] = 0 \tag{3}$$

Advice: You want to write explicitly what the definition of w as the best linear regression under p(x, y) implies, in terms of minimizing the expected loss (which, to remind you, is an integral w.r.t. x and y) similarly to how we treated the empirical loss in the case of $\widehat{w}$.

Let us begin by defining true risk as a function of the parameter w such that $R(w) = E_{(x_o,y_o)\ p(x,y)}[\ell(f(x_o,w),y_o)]$ where $\ell(f(x_o,w),y_o)$ is the loss function. We can expand this expectation in terms of the marginal densities of x_o and y_o such that, $R(w) = E_{x_o\ p(x)}[E_{y_o\ p(y|x)}[\ell(f(x_o,w),y_o)|x_o]$. By the definition of expectation, we can rewrite this as $\int_{x_o} E_{y_o\ p(y|x)}[\ell(f(x_o,w),y_o)|x_o]$. For the case of linear regression, assume a least squares loss function, $\ell(f(x,w),y) = \sum_{i=1}^{N}(y - w\cdot x_i)^2$. Now, to minimize the true risk, we must pick the value of w that minimizes the inner conditional expectation, which is equivalent to picking the optimal value of w, w^*, such that $\ell(f(x,w),y)$ is minimized. Now, we get that $R(w) = \int_{x_o} E_{y_o\ p(y|x)}[\sum_{i=1}^{N}(y - w\cdot x_i)^2|x_o]p(x_o)dx_o$. However, can give this in matrix product form, $R(w) = \int_{x_o} E_{y_o\ p(y|x)}[(y - w\cdot x)^2|x_o]px_o dx_o$. Now, we can differentiate $R(w)$ with respect to w: $\dfrac{\partial R(w)}{\partial w} = \dfrac{\partial}{\partial w}\int_{x_o} E_{y_o\ p(y|x)}[(y - w^Tx)^2|x_o]px_o dx_o$. Since we showed that we must only minimize the inner conditional to minimize $R(w)$, we can push the differentiation inside the integral.

$$\frac{\partial}{\partial w}E_{y\ p(y|x)}[(y - w_*^Tx)^2|x_o] = 0$$

$$-2E_{y\ p(y|x)}[(y - w_*^Tx)x|x_o] = 0$$

By definition, we stated earlier that w_* is the optimal value of w that sends the predictive error to 0, so we see that the mean of the predictive errors is equal to 0 for $\hat{w} = w_*$. Thus, if we take any linear function of the input x, by linearity of expectation, a^Tx would still be uncorrelated to the predictive error. Thus,

$$E_{y\ p(y|x)}[(y - w_*^Tx)a^Tx] = 0$$

Problem 2 [5 points] Explain (succinctly but precisely) how the original statement, regarding zero correlation between the prediction errors made by $\widehat{\mathrm{w}}$ estimated by least squares with any linear function of the training $\{\mathrm{x}_i\}$, follows from (3).

It follows from the result of 3 that the predictive error has a covariance of 0 with any linear function of x, since the definition of $Cov(U, V) = E[UV] - E[U]E[V]$. We know that if we model the predictive error as a random variable U, and linear functions of x as V, then we have shown already that $E[UV] = 0$.

Next, we consider the effect of scaling the input in the regression problem. This becomes relevant, for instance, in polynomial regression with high order models, to prevent very large or very small values and the resulting potential for numerical instability (think about the case of $x = 0.01$ or $x = 1000$ for 10-order model...). It seems that a good solution could be to multiply each column of the design matrix X by a suitable number so that the range of that column (i.e. of the corresponding dimension in the regression input space) be fixed, say, to $[-1, 1]$.

Suppose that the data are scaled by multiplying the j-th dimension of the input by a non-zero number c_j. We will denote a single normalized data point by $\tilde{\mathrm{x}} \equiv [1, c_1x_1, \ldots, c_dx_d]^\top$ and the resulting design matrix by $\tilde{\mathrm{X}}$.

Problem 3 [20 points] Let $\widehat{\mathrm{w}}$ be the least squares estimate of the regression parameters from the unscaled X, and let $\tilde{\mathrm{w}}$ be the solution obtained from the scaled $\tilde{\mathrm{X}}$. Show that the scaling does not change optimality, in the sense that $\widehat{\mathrm{w}} \cdot \mathrm{x} = \tilde{\mathrm{w}} \cdot \tilde{\mathrm{x}}$
Hint: You may find it helpful to express the scaling as a linear operator, yielding a matrix-product expression, and to equip yourself with a matrix reference, such as the Matrix Cookbook (look it up if you are not already familiar with it!)

Let's begin by defining the scaling operator as a square diagonal matrix such that the normalized data point $\bar{x} = Cx$ and the entire normalized design matrix is equal to $\bar{X} = XC$. Now, let us define the loss function for the new normalized least squares estimate $L(\tilde{w}, \tilde{X}, y) = \frac{1}{N}(y - \tilde{X}\tilde{w})^T(y - \tilde{X}\tilde{w}) = \frac{1}{N}(y^T - \tilde{w}^T\tilde{X}^T)(y - \tilde{X}\tilde{w})$. Now, as usual, differentiate with respect to w to find the optimal $\tilde{w}$ that minimizes the loss function.

$$\frac{\partial L(\tilde{w}, \tilde{X}, y)}{\partial w} = \frac{1}{N}\frac{\partial}{\partial w}[y^T y - y^T\tilde{X}\tilde{w} - \tilde{w}^T\tilde{X}^T y + \tilde{w}^T\tilde{X}^T\tilde{X}\tilde{w}]$$

$$0 = \frac{1}{N}[0 - 2\tilde{X}^T y + 2\tilde{X}^T\tilde{X}\tilde{w}]$$

$$\tilde{X}^T\tilde{X}\tilde{w} = \tilde{X}^T y$$

$$\tilde{w} = (\tilde{X}^T\tilde{X})^{-1}\tilde{X}^T y = [(XC)^T(XC)]^{-1}(XC)^T y$$

Since C is diagonal, it is also symmetric so $C^T = C$. Also, $X^T X issymmetric$

$$\tilde{w} = (CX^T XC)^{-1}CX^T y = (C^2(X^T X))^{-1}CX^T y$$

$$\tilde{w} = C^{-1}(X^T X)^{-1}X^T y = C^{-1}\widehat{w}$$

$$\tilde{w} \cdot \tilde{x} = \widehat{w} \cdot x = C^{-1}\widehat{w} \cdot Cx$$

$$C^{-1}\widehat{w} \cdot Cx = \widehat{w} \cdot x$$

$$w^T C^{-1}Cx = \widehat{w} \cdot x$$

$$w^T X = \widehat{w} \cdot x$$

When the dimensionality d of the data (aka the number of features) is much larger than the number of training instances n (i.e. $d \gg n$), the matrix $\mathrm{X}^\top \mathrm{X}$ is not full rank and thus can not be inverted. Therefore, instead of minimizing the least squares loss, we minimize the following loss function:

$$\widehat{R}_\lambda(\mathrm{w}) = \sum_{i=1}^{n} (y_i - \mathrm{w}^\top \mathrm{x}_i)^2 + \lambda \sum_{j=1}^{d} w_j^2 = (\mathrm{y} - \mathrm{X}\mathrm{w})^\top (\mathrm{y} - \mathrm{X}\mathrm{w}) + \lambda \|\mathrm{w}\|_2^2. \tag{4}$$

Problem 4 [20 points] Find $\widehat{\mathrm{w}}$ that minimizes the empirical risk in equation (4).

$$\widehat{R}_\lambda(w) = (y - Xw)^T(y - Xw) + \lambda\|w\|\|_2^2$$

$$\widehat{R}_\lambda(w) = (y^T - w^T X^T)(y - Xw) + \lambda\|\|w\|\|_2^2$$

$$\widehat{R}_\lambda(w) = y^T y - 2(X^T w^T y) + w^T X^T Xw + \lambda\|w\|_2^2$$

$$\widehat{R}_\lambda(w) = y^T y - 2(X^T w^T y) + w^T X^T Xw + \lambda\|w\|_2^2$$

$$\frac{\partial \widehat{R}_\lambda(w)}{\partial w} = 0 - 2(X^T y) + 2X^T X\widehat{w} + 2\lambda\widehat{w} = 0$$

$$0 = -(X^T y) + X^T X\widehat{w} + \lambda\widehat{w}$$

$$(X^T y) = X^T X\widehat{w} + \lambda\widehat{w}$$

$$(X^T y) = \widehat{w}(X^T X + \lambda)$$

$$(X^T y)(\lambda + X^T X)^{-1} = \widehat{w}$$

II. Asymmetric loss

In this section, we will consider a regression model associated with a different loss function.

In class we have discussed the squared loss

$$\ell(\hat{y}, y) = (\hat{y} - y)^2, \tag{5}$$

which is symmetric – the sign of the prediction error in by with respect to ground truth y doesn't matter. In some applications, this may not be a reasonable loss. For instance, consider the problem of estimating pressure necessary to put in the tire of a truck as a function of some parameters of the road, environment, vehicle etc. Putting pressure which is too low could reduce performance; if it's extremely low it can of course be dangerous. However, putting too much pressure may quickly result in catastrophic damage to the vehicle.

In this situation, we would like to penalize for positive errors differently (more severely) than for the negative errors of the magnitude. We can express this in the following, asymmetric squared loss:

$$\ell_\alpha(\hat{y}, y) = \begin{cases} (\hat{y} - y)^2 & \text{if} \quad \widehat{y} \leq y, \\ \alpha(\hat{y} - y)^2 & \text{if} \quad \widehat{y} \geq y. \end{cases} \tag{6}$$

The coefficient α determines how much more we worry about over-predicting y than about under-predicting. We will assume that $\alpha \geq 1$. See Figure 1 for an illustration.

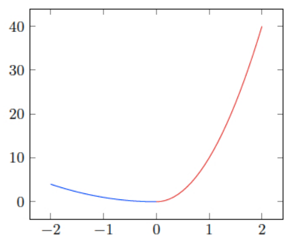


Figure 1: Plot of asymmetric loss ℓ_{10}; blue portion corresponds to negative errors (under-prediction), red to positive errors.

For the following problems, we have provided you with three data sets: training, validation and test, drawn from the same (unknown to you) $p(\mathrm{x}, y)$. The data are in a HDF5 file `data.h5`. You will use training set to fit different models; validation to select between models, and test set to report the accuracy with the model that you end up choosing. That means that the test set will be only used once, to compute and report test loss with the model of choice.

For your convenience, we provide most of the code you will need in the experiments below, with a few missing pieces that you will need to fill in. If you are new to Python, don't worry too much about efficiency and elegance of your code, although those are of course good goals eventually; for now just make sure it works correctly and is readable.

The code "skeleton", with quite a bit of flesh on it, is provided as a Jupyter notebook `PS1-asymloss-notebook.ipynb`, along with an auxiliary file `utils.py` containing a couple of functions you will need; take time to read and understand the provided code, especially if you are new to Python.

Please submit the notebook with all the code filled in and with the output of your experiments, after the name change as stated in the preamble of this problem set. Please write your comments, explanations and interpretation of the results as requested in the problems below, in the PDF solution file.

Problem 5 [15 points]

Fit linear, quadratic and cubic models to the training data under the standard (symmetric) squared loss (5). Visualize the fit under the three models . To make it easier to parse your figures, let's agree on the color scheme: the plots will be blue (linear), green (quadratic) and red (cubic) solid lines.

Evaluate, for each of the three models, the (symmetric) squared loss and the log-likelihood of the training data under the Gaussian noise assumption.

Note: you will need to derive ML estimate for the Gaussian noise variance σ^2, in addition to the ML estimate for the model parameters w; write the missing code snippet for this, and explain in the solution file how you derived it.

Finally, compute and report (symmetric) squared loss for each of the three models on the validation set. Comment on the relative performance of the models, and on the gap, if any, between validation and training loss values. Select one of the three models based on this evaluation, and explain your selection. Let's call the model you end up choosing here model A.

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

Empirically, it was found that a model of degree 2 has the lowest validation error but slightly higher training loss and variance, and lower log-likelihood (a higher log-likelihood is better since maximizing log-likelihood minimizes log-loss). These numbers make sense since without cross validation, the model that is chosen will always be more complex since it will always have the lowest training loss without fail. However, the model of degree 2 has the lowest validation loss, meaning it performs the best in terms of generalizing out of the three, so we choose model A to be of degree 2. For the MLE for Gaussian noise variance, we know that the noise is a 0 mean Gaussian random variable, so the derivation process is simply, which ultimately just leads us to the fact that the estimator for Gaussian noise is the sample variance.

Problem 6 [25 points]

Now we want to repeat the experiment above but under the asymmetric loss function (6). Since there is no closed form solution, we will need to rely on gradient descent. Derive the expression for the gradient of the loss (6) with respect to the parameter vector w, for a general form of linear regression (i.e., with inputs represented by a feature vector ϕ(x), which could capture linear, quadratic or cubic model).

Complete the provided code for gradient descent using the derivation for the gradient, and run it to fit linear, quadratic and cubic models under asymmetric squared loss. Visualize the resulting three models (use same color scheme as before, but dashed lines) and report the asymmetric squared loss values and the log-likelihood of the model *under the standard Gaussian noise model* on training data, and the asymmetric squared loss on validation data. Compare these numbers to those you obtained with symmetric loss, and comment on the relative performance of the three models, as well as the performance of the equal degree models across the two loss functions.

Next, using the outcome of your experiment, select a model among the three (linear / quadratic / cubic) and explain your selection. Let's call the resulting model B.

Evaluate model A (selected with symmetric loss) and model B on test set, and interpret your findings; which model do you conclude to be a better choice based on this entire experiment? Choice of evaluation metrics is up to you, and you should motivate it succinctly but clearly.

Hint: You may want to start by reviewing the derivation of the gradient of squared loss, and extend it to asymmetric loss.

degree 1:
train loss 0.456437
val loss 0.374340
sigma^2: 0.159223
log-likelihood -1.005706

degree 2:
train loss 0.089027
val loss 0.076941
sigma^2: 0.036509
log-likelihood -0.122804

degree 3:
train loss 0.088523
val loss 0.078923
sigma^2: 0.036320
log-likelihood -0.114172

Here, based on a set of the same training data, we found that a degree 2 polynomial using an asymmetric loss model yields the lowest validation error, so we should use this as our model for the entire experiment.

PS1-asymloss-notebook

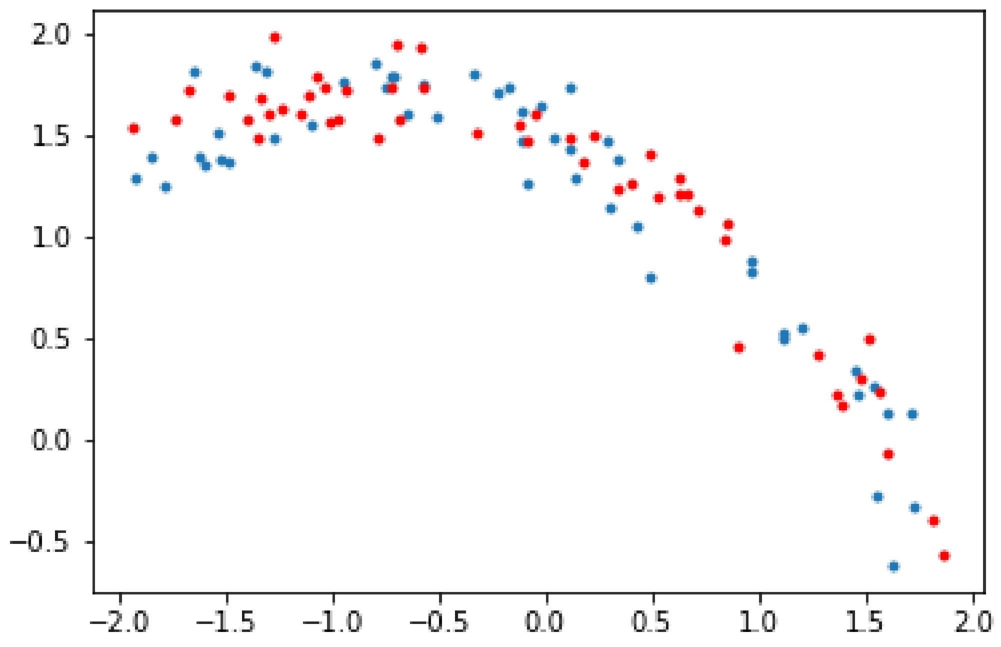
February 27, 2017

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 truth

             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
             -------
```

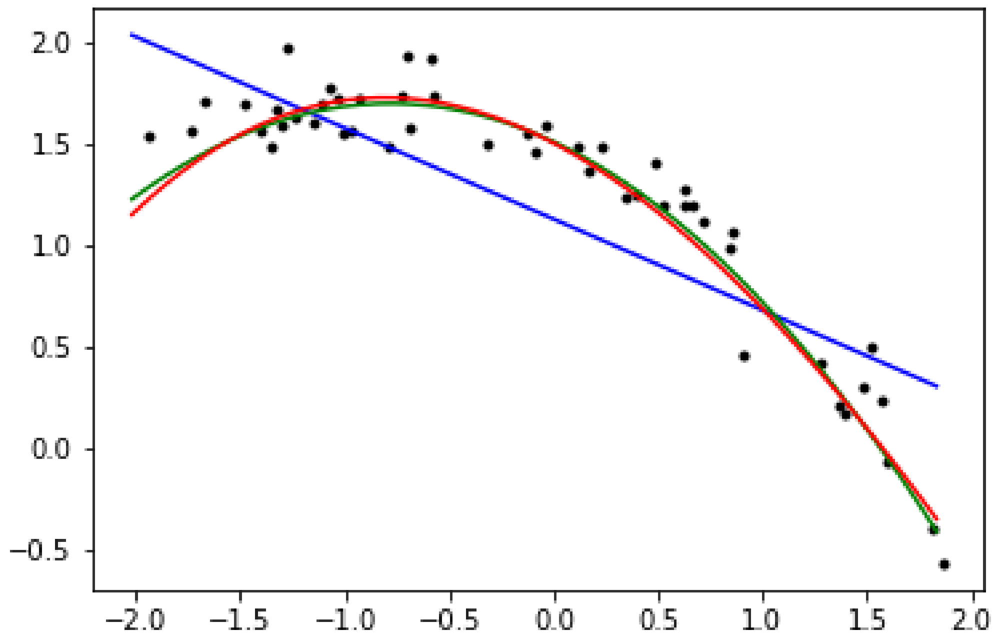
```
            simga2 : a scalar
                The estimated variance (sigma squared)
            loglike : a scalar
                The log-likelihood under the Gaussian noise model N(0,sigma2)
            """
            N = X.shape[0]    # number of rows in X
            # now estimate the variance 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])
```

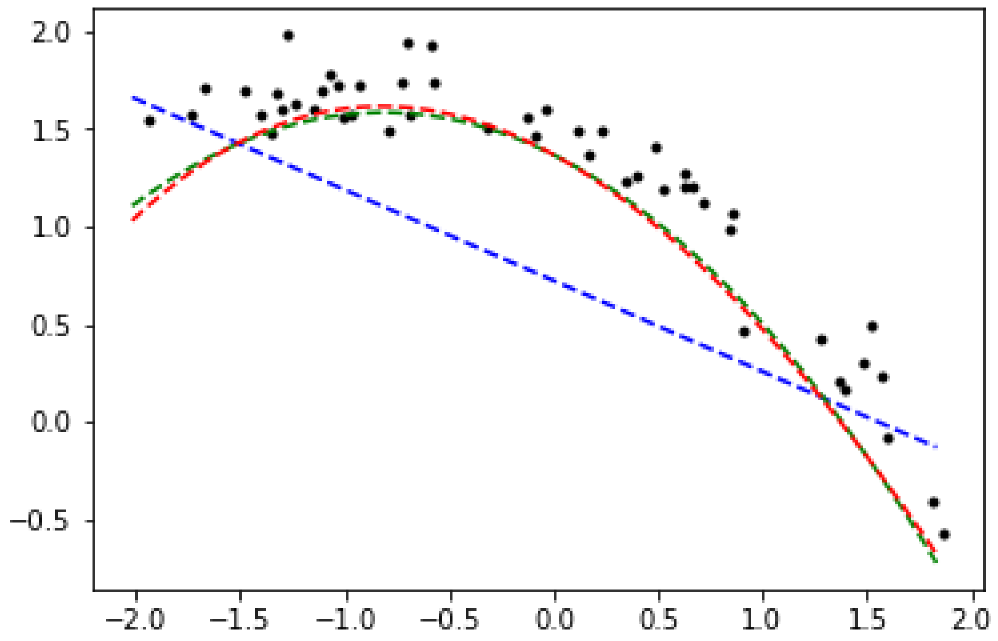
Now we can fit different models, and evaluate their performance on train and val

```
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 [ ]:
```