Comp 6321 - Machine Learning - Assignment 2

Federico O'Reilly Regueiro

October 18th, 2016

Question 1:

1.a Partition the data into training / testing, 90 to 10 and perform and plot $L2$ regularization

The data was partitioned pseudo-randomly[1].

```
phi = load('hw2x.dat');
phi = [phi, ones(size(phi,1),1)];
y = load('hw2y.dat');

% Partition the data randomly.
idxs = randperm(size(phi, 1));
idx_train = idxs(1:89);
idx_test = idxs(90:99);

phi_train = phi(idx_train, :);
y_train = y(idx_train);
phi_test = phi(idx_test, :);
y_test = y(idx_test);
```

Next, a range of lambdas was chosen, going from 0 to almost 125000 in order to get a good sense of the trend of both the error and the coefficients. The former was plotted for a range of small values as well as along the whole set of lambdas for which the RMS error was calculated. Two plots were done in order to allow us to see the behaviour of the test and training errors for lower values of λ as well as the overall trend of the RMS the resulting plot can be seen in Figure 1.

```
lambdas = 0:0.1:50;
lambdas = lambdas .^ 3;
```

[1]I have included the permutation indexes yielded by matlab for the instance of the 90 / 10 partition from which the plots and values were drawn. Suffice it to uncomment two lines and comment-out two others in order to partition the data randomly, yet similar results (at different scales of λ) can be observed.

```
w = zeros(length(lambdas), size(phi,2));

for lambda = lambdas
  idx = lambda == lambdas;

  % Train the model
  w(idx, :) = pinv(phi_train' * phi_train ...
              + (lambda * eye(size(phi_train, 2)))) ...
              * (phi_train' * y_train);

  h_phi_train(:, idx) = phi_train * w(idx, :)';
  j_h_train(idx) = rms(h_phi_train(:, idx) - y_train);

  % Now compare to the test
  h_phi_test(:, idx) = phi_test * w(idx, :)';
  j_h_test(idx) = rms(h_phi_test(:, idx) - y_test);
end
```

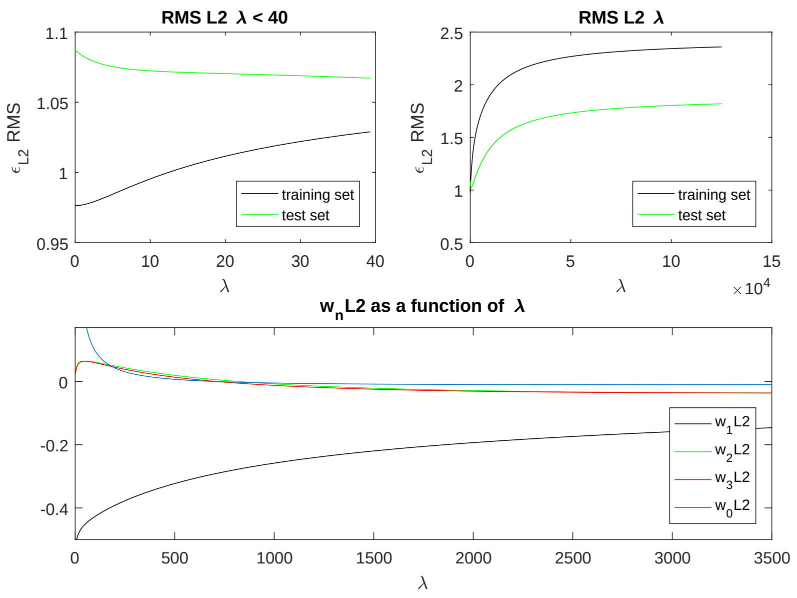

Figure 1: Plot of the RMS training and testing error as well as the coefficients over a wide range of λ.

On the top-left plot of figure 1 can observe that as expected, the test error goes down as λ grows. Howerver, as λ continues to increase, the test error closely follows the training error as they both grow since the restrictions on the coefficients make for a worse fit at a certain point.

1.b Use the `quadprog` function in Matlab for $L1$ regularization

In order to use `quadprog` for regularization, we must first find the Hessian matrix `H` as well as other parameters `f`, `A`, `b` in the specific format that Matlab requires.

We recall that for $L1$ regularization the expression we must minimize is:

$$\arg\min_{w} \frac{1}{2}(\boldsymbol{\Phi w} - \boldsymbol{y})^T(\boldsymbol{\Phi w} - \boldsymbol{y}) + \frac{\lambda}{2}\sum_{k=0}^{K-1}|w_k| \tag{1}$$

Which is equivalent to finding:

$$\begin{gathered}\arg\min_{w}(\boldsymbol{\Phi w} - \boldsymbol{y})^T(\boldsymbol{\Phi w} - \boldsymbol{y}) \\ \sum_{k=0}^{K-1}|w_k| \leq \eta\end{gathered} \tag{2}$$

And expands to:

$$\begin{gathered}\arg\min_{w}\boldsymbol{w}^T\boldsymbol{\Phi}^T\boldsymbol{\Phi w} - 2y^T\boldsymbol{\Phi w} \\ \sum_{k=0}^{K-1}|w_k| \leq \eta\end{gathered} \tag{3}$$

And for which we can remove the constant term $\boldsymbol{y}^T\boldsymbol{y}$, yielding:

$$\begin{gathered}\arg\min_{w}\boldsymbol{w}^T\boldsymbol{\Phi}^T\boldsymbol{\Phi w} - \boldsymbol{2y}^T\boldsymbol{\Phi w}^T \\ \sum_{k=0}^{K-1}|w_k| \leq \eta\end{gathered} \tag{4}$$

Matlab's `quadprog(H, f, A, b)` function, gives the optimal $\boldsymbol{x}$ corresponding to the expression $\arg\min_{x} \frac{1}{2}\boldsymbol{x}^T\boldsymbol{Hx} + \boldsymbol{f}^T\boldsymbol{x}$, subject to constraints $\boldsymbol{Ax} \leq \boldsymbol{b}$. We can thus take `H :=` $2\boldsymbol{\Phi}^T\boldsymbol{\Phi}$, then `f :=` $-2\boldsymbol{y}^T\boldsymbol{\Phi}$, `A :=`$\boldsymbol{P}$, where for a system with n variables, $\eta\boldsymbol{P}$ is the matrix with 2^n permutations of $[b_1, b_2, \dots b_n], b \in \{-1, 1\}$ and lastly `b :=` $c\ \overrightarrow{\mathbf{1}}$, where $\overrightarrow{\mathbf{1}}$ is an all-one vector of length 2^n that places an upper bound, η, to the expression $\sum_{k=0}^{K-1}|w_k|$, such that $\sum_{k=0}^{K-1}|w_k| \leq \eta$. We note that η is roughly equivalent to $\frac{1}{\lambda}$ which we use so that we may compare the effects of $L1$ and $L2$ regularizations on a similar range of values.

Thus we end up with the following code:

```
etas = 1./lambdas;
for eta = etas
  idx = eta == etas;
  w_quad(idx,:) =
      quadprog(
          2*(phi_train'*phi_train), ...
         -2*(phi_train'*y_train), ...
```

```
                [ 1, 1, 1, 1;    1, 1,-1, 1; ...
                   1,-1, 1, 1;    1,-1,-1, 1; ...
                  -1, 1, 1, 1;   -1, 1,-1, 1; ...
                   1,-1, 1, 1;   -1,-1,-1, 1; ...
                   1, 1, 1, -1;    1, 1,-1, -1; ...
                   1,-1, 1, -1;    1,-1,-1, -1; ...
                  -1, 1, 1, -1;   -1, 1,-1, -1; ...
                   1,-1, 1, -1;   -1,-1,-1, -1], ...
            eta * [1; 1; 1; 1; 1; 1; 1; 1]);

    h_phi_train_quad(:, idx) =
                phi_train * w_quad(idx, :)';
    j_h_train_quad(idx) =
                rms(h_phi_train_quad(:, idx) - y_train);

    % Now validate
    h_phi_test_quad(:, idx) =
              phi_test * w_quad(idx, :)';
    j_h_test_quad(idx) =
              rms(h_phi_test_quad(:, idx) - y_test);
end
```

1.c Plot $L1$ RMS and coefficients, w against λ and comment

Although it is not exactly equivalent, we shall simplify the comparison of $L1$ and $L2$ throughout this section by using $\lambda \approx \frac{1}{\eta}$ as it gives a clearer idea. In Figure 2, we can again notice how the test error slightly decreases for the lowest values of $\approx \lambda$ and then monotonically increases as the coefficients are all forced towards 0. We also note that $L1$ yields a lower minimum error (1.0098) than $L2$ regularization (1.0257).

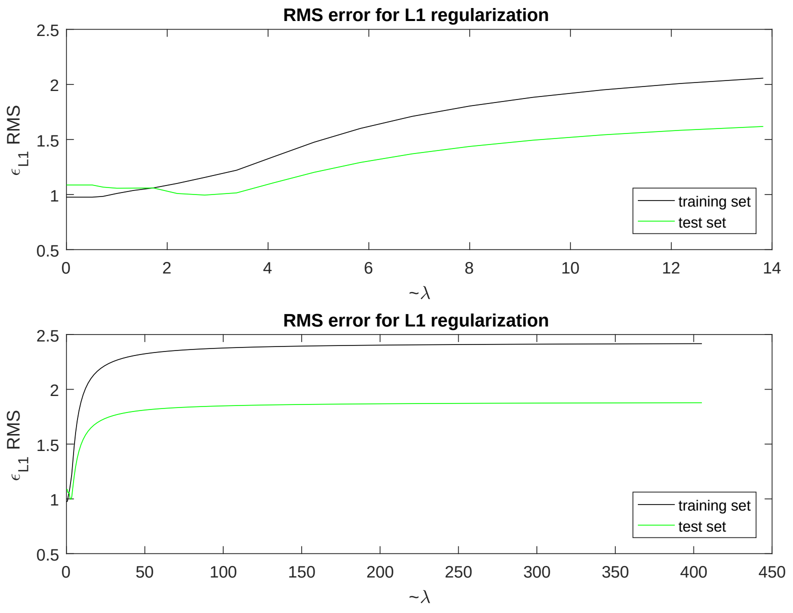

Figure 2: The RMS training and testing error for a wide range of $\frac{1}{\eta} \approx \lambda$.

By the same token, in figure 3 we can see how both w_2 and w_3 sharply decrease to 0 when $\approx \lambda = 2$ and the model relies solely on w_1[2] which then decreases gradually, as opposed to figure 1, where we can observe how all coefficients approach 0 at a similar rate during $L2$ regularization. Conversely and as expected, we can see that both errors and coefficients are equal between $L1$ and $L2$ regularization when $\lambda = 0$.

This particular data-set would lead to believe that the data was generated mainly by some function $f(w_1)+\epsilon$. This hypothesis is supported by the following output:

```
corr(y, phi(:,1:3))
ans =
   -0.848189    -0.017339    -0.024943
```

which reveals that the correlation between $\Phi_{:,1}$ and y is much larger than between other columns of Φ and y.

[2] w_0 is the bias term, so we can't really say the model relies on it a it is not an input.

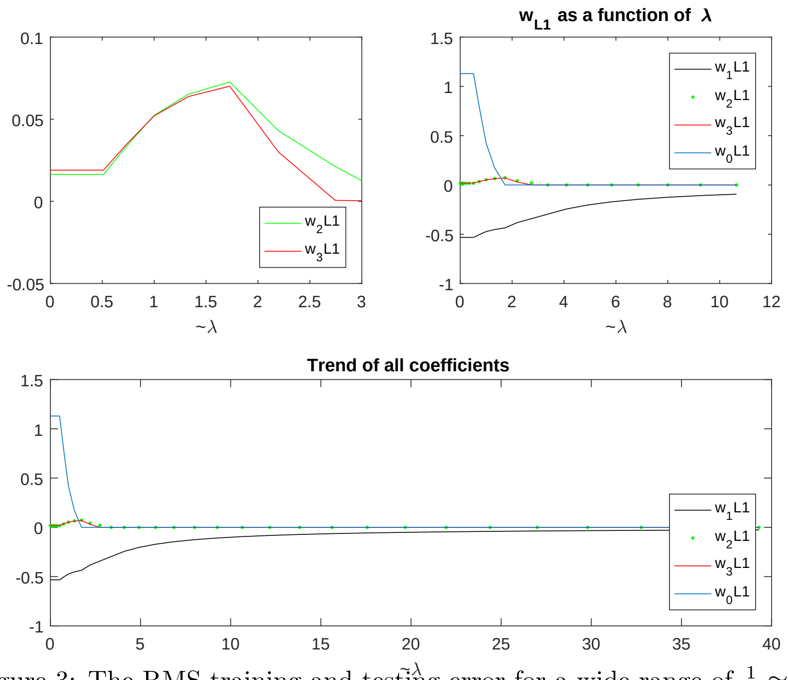

Figure 3: The RMS training and testing error for a wide range of $\frac{1}{\eta} \approx \lambda$.

Question 2: Dealing with missing data, fill in $x_{i,n}$ with class-conditional means?

First, we write $\mu_{c,i}$ to represent $E(x_i|y = c)$ and we assume independence between features. Then, since our classifier is Gaussian, we know that $P(x|y = 1)$ and $P(x|y = 0)$ are modelled as follows:

$$P(x|y = c) = \frac{1}{\sqrt{2\pi|\mathbf{\Sigma}|}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu_c})^T \mathbf{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu_c}), \quad c\in\{0,1\}} \tag{5}$$

We turn our attention to the numerator of the exponent, which can also be written in the following manner:

$$\sum_{i=1}^{n}[(x_i - \mu_{c,i}) \sum_{j=1}^{n}(\Sigma^{-1}{}_{i,j}(x_j - \mu_{c,j})] \tag{6}$$

For the value of a given feature n, replaced by its class-conditional mean, $\mu_{c,n}$, this expression becomes 0; we further develop this by analyzing the log-odds:

$$log\frac{P(y = 1|\boldsymbol{x})}{P(y = 0|\boldsymbol{x})} = log\frac{P(y = 1)}{P(y = 0)} + log\frac{\frac{1}{\sqrt{2\pi|\mathbf{\Sigma}|}}e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu_1})^T\mathbf{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu_1})}}{\frac{1}{\sqrt{2\pi|\mathbf{\Sigma}|}}e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu_0})^T\mathbf{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu_0})}} \tag{7}$$

Since the matrix sigma is shared, we can write:

$$log\frac{P(y=1|\boldsymbol{x})}{P(y=0|x)} = log\frac{P(y=1)}{P(y=0)} + log\frac{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu_1})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu_1})}{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu_0})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu_0})} \tag{8}$$

Then we expand the exponent:

$$log\frac{P(y=1|\boldsymbol{x})}{P(y=0|\boldsymbol{x})} = log\frac{P(y=1)}{P(y=0)} + log\frac{-\frac{1}{2}\sum_{i=1}^{n}[(x_i-\mu_{1,i})\sum_{j=1}^{n}(\Sigma^{-1}{}_{i,j}(x_j-\mu_{1,j})])}{-\frac{1}{2}\sum_{i=1}^{n}[(x_i-\mu_{0,i})\sum_{j=1}^{n}(\Sigma^{-1}{}_{i,j}(x_j-\mu_{0,j})])} \tag{9}$$

Where we can clearly see that the contribution of x_n does not change the ratio of the log-odds given by all other features, since we have choosen $x_n = \mu_{c,n}$ for $c \in \{0,1\}$ where $\mu_{c,n}$ is the class-conditional means for class c

Question 3: Naive Bayes assumption, suppose a feature gets repeated in the model

Question 4: If only anything like this had been touched-upon in class...

Question 5: Implementation of Logistic Regression and Gaussian Naive-Bayes

A little bit about our data

Before plunging into the implementation task at hand, we take a look at some of the defining characteristics of our data set. With this heuristic purpose in mind, a routine was created to visualize some characteristics of our dataset. The code is included in the `plot_dyn.m` script.

```
for = 1:32
    figure(3)
    plot(X(y==1, i), zeros(sum(y==1),1), 'g.')
    hold on
    plot(X(y==0, i), zeros(sum(y==0),1), 'r.')
    exes = min(X(:,i))-1:0.1:max(X(:,i))+1;
    plot(exes, normpdf(exes, mean(X(y == 1, i)),
              std(X(y == 1, i))), 'g')
    plot(exes, normpdf(exes, mean(X(y == 0, i)),
              std(X(y == 0, i))), 'r')
    title(['feature ', num2str(i), ' class distributions'])
    hold off
```

```
        for j = i+1:32
            figure(4);
            plot(X((y == 1),i), X((y == 1),j), 'g+');
            hold on;
            plot(X((y == 0),i),X((y == 0),j), 'r.');
            hold off;
            title(['feature␣', num2str(i),
                   'against␣feature␣', num2str(j)]);
            pause(0.1)
        end
    end
```

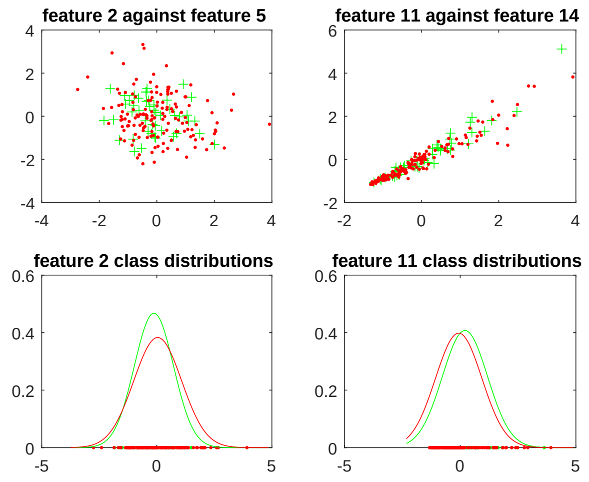

Figure 4: Some of the feature-pairs do not show a strong correlation (top-left), while other pairs (top-right) display a clear correlation. The class-distributions for single features are fairly close in most cases (bottom).

Firstly, we observe in figure 4 that the class-means and class variances of each one of our features are rather close, we have picked two examples of scatter-plots for feature pairs and two examples for the per-feature class-distribution. We can also observe that the data is nicely centered aroud 0 for all features and since the variances are not too different between all features[3], then we can safely say that logistic regression would not benefit much from feature normalization.

A second characteristic that becomes quickly obvious is that some features are highly correlated, an example of this is plainly visible in figure 4 and we can

[3]the avid reader is urged to run the included `plot_dyn.m` to cross-check this fact

also see this well represented in by the fact that the variance-covariance matrix of the inputs in figure 5 has large fairly values outside of the diagonal entries; this is not ideal for the naive assumption made by the gaussian-naive-bayes model.

Lastly, we note that there are 46 entries corresponding to class 1 out of 194 total entries. So if we were to arbitrarily classify all of our test data as negative, we would have a 76% rate of succesful classification.

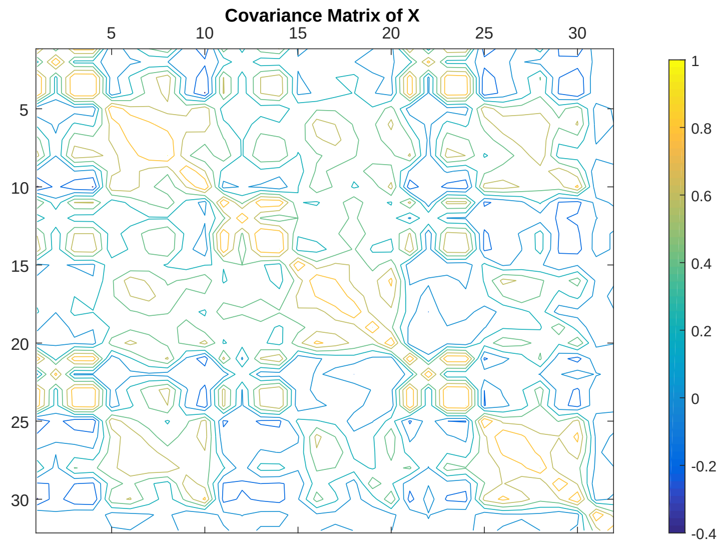

Figure 5: The variance-covariance matrix of the inputs shows large values outside of the diagonal entries.

5.a Logistic rergession

For logistic regression, we chose to carry out a gradient descent implementation. The implementation is contained in the `LR_grad.m` function file with an auxiliary `find_alpha.m` function file.

Since gradient descent may get stuck on shallow local minima, we chose to iterate the learning process over a randomly-picked collection of 24 weight vectors, aiming at keeping the one that should yield the lesser error after all iterations of gradient descent should be done.

`LR_grad.m`:

```
function [ w, w_inits ] =

    % default to 24 random starting points
    if nargin < 4 || isempty(num_rand_inits)
```

```
        num_rand_inits = 24;
    end

    % if no random starting points are given, make some
    if nargin < 3 || isempty(w_inits)
        w_inits = rand(size(X,2), num_rand_inits);
    else
        num_rand_inits = size(w_inits(2));
    end

    w = w_inits;
    for i = 1: num_rand_inits
        done = false;
        prev_epsilon = realmax * ones(length(y),1);
        iterations = 1;
        % find a good learning rate for each vector
        lr = find_alpha(X, y, w(:,i));
        % Don't allow GD to run after it's found minima
        % nor for too long
        while ~done && (iterations < 2000)
            epsilon = y - (1./(1+exp(-X*w(:,i))));
            % if our prediction error is very small
            % or it's not really changing, we're done
            if (abs(sum(epsilon)) < .1) ...
                    || abs(sum(epsilon) ...
                      - sum(prev_epsilon)) < 0.01
                done = true;
            else % keep going down
                w(:,i) = w(:,i) + lr * (X'*epsilon);
                prev_epsilon = epsilon;
            end
            iterations = iterations + 1;
        end
    end
end
```

For each initial vector, an optimal learning rate is chosen from $[\frac{1}{1}, \frac{1}{2}, \ldots, \frac{1}{10}]$. The learning rates were left fixed, as implementing the Robbins-Monro conditions lead to a substantial degradation in performance - speedwise. The rationale to support this choice being that as the gradient approaches 0, the update to the $\boldsymbol{w}$ vector will inherently be smaller.

The method for finding an optimal learning rate is implemented in the function `find_alpha.m`:

```
function alpha = find_alpha(X, y, w)
    lrs = 1./(1:10);
    errors = zeros(1, length(lrs));
```

```
        for i = 1: length(lrs)
            iterations = 1;
            lr = lrs(i);
            while iterations < 100
                epsilon = y - (1./(1+exp(-X*w)));
                w = w + lr * (X'*epsilon);
                iterations = iterations + 1;
            end
            errors(i) = sum(epsilon);
        end
        [dummy, idx] = min(abs(errors));
        alpha = lrs(idx);
    end
```

In order to perform 10-fold cross-validation, we used matlab's `cvpartition` function, which yields a homogenous grouping of entries per fold[4]. Matlab's `randperm` was also used to chose the entries randomly but without repetition for each fold. The following code from `A2_q5_driver.m` conatins the calls to `LR_grad.m`[5].

```
        ...
    folds_info = cvpartition(length(y), 'kfold', num_folds);
    folds_idx = randperm(length(y));

    num_rand_inits = 24;
    features = 1:33;

    for fold = 1:num_folds
        % just indexing for folds
        idxs_prev = 1:sum(folds_info.TestSize(1:(fold -1)));
        if ~isempty(idxs_prev)
            offset = idxs_prev(end);
        else
            offset = 0;
        end
        idxs_xcl = (1:folds_info.TestSize(fold))+offset;
        idx_after_skip = length(y) -
              (sum(folds_info.TestSize((fold + 1):end)) - 1);
        idxs_next = idx_after_skip:length(y);
        X_train = X(folds_idx([idxs_prev, idxs_next]),
                    features);
        X_test = X(folds_idx(idxs_xcl), features);
        y_train = y(folds_idx([idxs_prev, idxs_next]));
        y_test = y(folds_idx(idxs_xcl));
```

[4]avoiding the final instance of a fold with 4 entries.

[5]The full version also contains calls to the naive Bayes Gaussian classifier, which will appearlater on.n section 5.b

```
        % Here's the actual call to logistic regression
        [w, w_init] = LR_grad(X_train, y_train, [],
                              num_rand_inits);
        w_grad(fold, :, :) = w;
        for i=1:num_rand_inits
            errors_grad(fold, i) =
                   sum(y_test ~= round(1
                   ./ (1 + exp( - X_test*w(:,i)))));
        end
    end
      ...
```

The choice of error was made so as to allow a proper comparison with the errors given by the Gaussian naive Bayes, or GNB hereafter. Since for GNB we used log-odds, the translation to a cross-entropy type error becomes rather unwieldy. We have thus take a cost function of the form:

$$J_w(x_i) = \begin{cases} 0 & \text{if } \hat{y_i} = y_i \\ 1 & \text{if } \hat{y_i} \neq y_i \end{cases}$$

This is easily done via the following matlab command[6]:

```
      min_err = min(abs(mean(errors_grad, 1)))}
```

5.b Implementation of GNB classifier

In order to setup our GNB algorithm, we separated the necessary actions into `gnb_train.m` and `gnb_predict.m`

The contents of `gnb_train.m`:

```
function [theta, mu_1, mu_0, Sigma] = gnb_train(X, y)
    theta = sum(y == 1)/length(y);
    idx_1 = y == 1;
    idx_0 = y == 0;
    mu_1 = mean(X(idx_1,:));
    mu_0 = mean(X(idx_0,:));
    x_to_mu = zeros(size(X));
    x_to_mu(idx_1,:) = X(idx_1,:)
                       - repmat(mu_1, sum(idx_1), 1);
    x_to_mu(idx_0,:) = X(idx_0,:)
                       - repmat(mu_1, sum(idx_0), 1);
    Sigma = (x_to_mu' * x_to_mu)./length(y);
end
```

And the contents of `gnb_predict.m`:

[6]Note that only the minimum error of the 24 logistic regressions performed is taken into account since this is one of the main reasons for having several random starting vectors.

```
function [log_odds, p_x_y_1, p_x_y_0] = ...
                gnb_predict(X, theta, mu_1, mu_0, Sigma)
    m = size(X, 1);
    n = size(X, 2);
    Sigma_inv = pinv(Sigma);
    norm_term = 1/((2*pi)^(n/2)*sqrt(det(Sigma)));
    X_to_mu_1 = (X - repmat(mu_1, m, 1));
    X_to_mu_0 = (X - repmat(mu_0, m, 1));

    alpha_1 =
            ((X_to_mu_1 * Sigma_inv * X_to_mu_1')
              .*eye(m)) * ones(m,1);
    alpha_0 =
             ((X_to_mu_0 * Sigma_inv * X_to_mu_0')
              .*eye(m)) * ones(m,1);
    p_x_y_1 = norm_term*exp(-alpha_1./2);
    p_x_y_0 = norm_term*exp(-alpha_0./2);
    log_odds = (alpha_0./2 - alpha_1./2) + ...
                repmat(log(theta/(1 - theta)),
                       length(alpha_1), 1);
end
```