Beta-Bernoulli Naïve Bayes

The data given are binarized with threshold 0, ie : $\mathbb{I}(x_{ij} > 0)$ which then are fitted into Beta-Bernoulli naïve Bayes model :

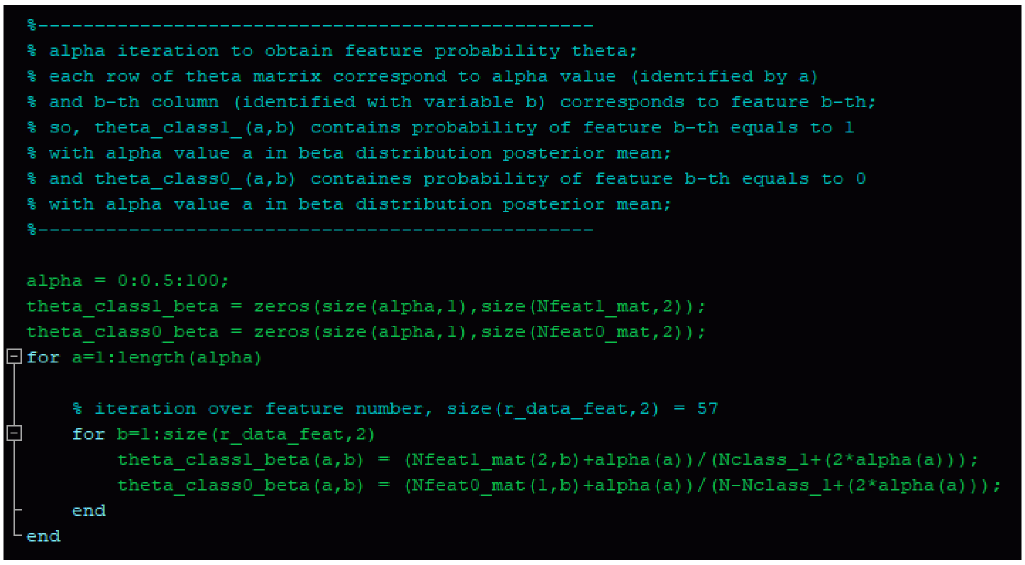
$$p(y = c|x, T) = \frac{p(y = c, x|T)}{p(x|T)} \propto p(y = c|T)p(x|y = c, T) = p(y = c|T) \prod_{j=1}^{D} p(x_j\,|y = c, T)$$

If our purpose is to find the class *c* that has highest value of $p(y = c|x, T)$, then the term $p(x|T)$ can be ‘dropped’ since its value doesn’t change with class (doesn’t contain class variable term *y*).

I will end up with term $p(y = c|T) \prod_{j=1}^{D} p(x_j\,|y = c, T)$ due to assumption from naïve bayes; features are conditionally independent, that is given class label *y* equals to *c*, term of joint distribution from a feature vector $p(x|y = c, T) = p(x_1, x_2, \dots x_D|y = c, T)$ can be computationally simplified into product of its marginals $\prod_{j=1}^{D} p(x_j\,|y = c, T)$ where D is the dimension of data (number of features).

In this exercise, feature distribution $p(x_j|y = c, T)$ is modelled with $Beta(\alpha, \alpha)$ distribution. Since the assignment paper requested to use Beta posterior predictive mean when computing $p(x_j|y = c, T)$, therefore term $p(x_j|y = c, T)$will be equal to :

$p(x_j = 1\big|y = c, T\big) = \bar{\theta}_{jc,1} = \frac{N_{jc}+a}{N_c+2a}$; where N_{jc} is the number of samples in class *c* whose feature *j*th is equal to 1 (as we know our feature are binarized). Oppositely, the probability of *j*th equals to 0 in class *c* is given by : $p(x_j = 0\big|y = c, T\big) = \bar{\theta}_{jc,0} = \frac{M_{jc}+a}{N_c+2a}$; where M_{jc} is the number of samples in class *c* whose feature *j*th is equal to 0. That should be in consistency with equation $M_{jc} + N_{jc} = N_c$ which makes sense since our feature are binarized and will only have two possible values, and when I add $p(x_j = 1\big|y = c, T\big) + p(x_j = 0\big|y = c, T\big) = \frac{N_{jc}+a+M_{jc}+a}{N_c+2a} = 1$. In the programming section, these feature probabilities are calculated along with various alpha $a = \{0, 0.5, 1, 1.5, 2, ..100\}$ which will result in two matrices with size 201-by-57 (201 from alpha size, and 57 from the number of features, and two matrices corresponds to possible feature values; 0 or 1). The details of the code is shown below:



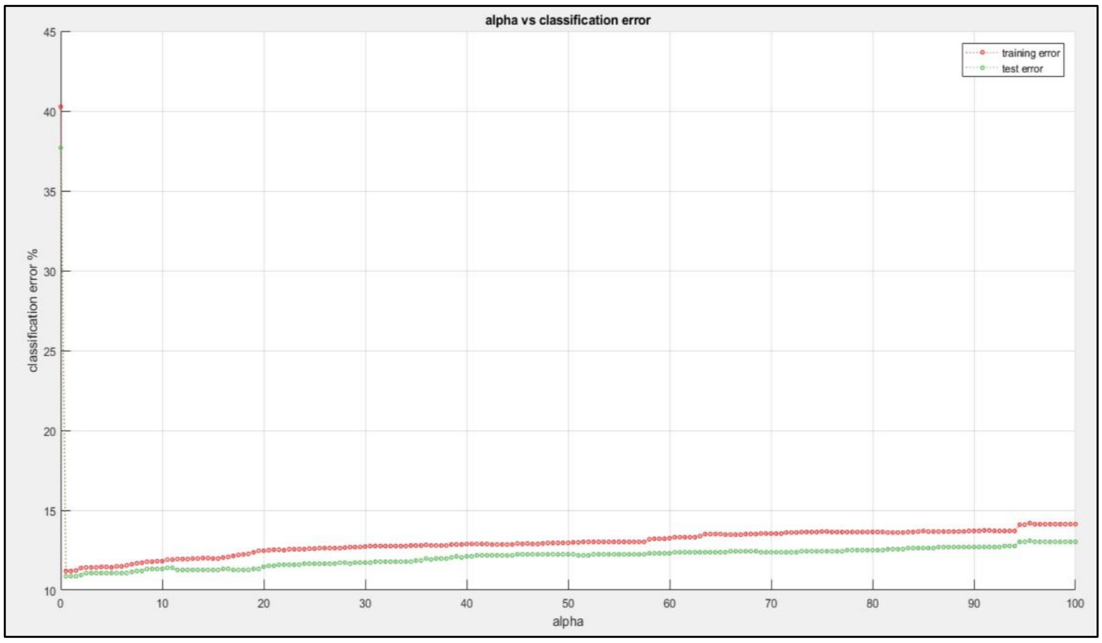
```
%------------------------------------------------
% alpha iteration to obtain feature probability theta;
% each row of theta matrix correspond to alpha value (identified by a)
% and b-th column (identified with variable b) corresponds to feature b-th;
% so, theta_class1_(a,b) contains probability of feature b-th equals to 1
% with alpha value a in beta distribution posterior mean;
% and theta_class0_(a,b) containes probability of feature b-th equals to 0
% with alpha value a in beta distribution posterior mean;
%------------------------------------------------

alpha = 0:0.5:100;
theta_class1_beta = zeros(size(alpha,1),size(Nfeat1_mat,2));
theta_class0_beta = zeros(size(alpha,1),size(Nfeat0_mat,2));
for a=1:length(alpha)

    % iteration over feature number, size(r_data_feat,2) = 57
    for b=1:size(r_data_feat,2)
        theta_class1_beta(a,b) = (Nfeat1_mat(2,b)+alpha(a))/(Nclass_1+(2*alpha(a)));
        theta_class0_beta(a,b) = (Nfeat0_mat(1,b)+alpha(a))/(N-Nclass_1+(2*alpha(a)));
    end
end
```

Here I need to compute both $p(x_j = 0\big|y = c, T\big) = \bar{\theta}_{jc,0}$ and $p(x_j = 1\big|y = c, T\big) = \bar{\theta}_{jc,1}$because I want to compare $p(y = 1|x, T)$ which is probability of x belongs to class 1 (spam) and $p(y = 0|x, T)$, x belongs to class 0 (non-spam). And we know the following :

- Spam probability (class y=1)
  - $p(y=1|x,T) \propto p(y=1|T)\prod_{j=1}^{D} p(x_j\,|y=1,T) = \pi_1 \prod_{j=1}^{D} p(x_j\,|y=1,\alpha)$
  - $\pi_1 \prod_{j=1}^{D} p(x_j\,|y=1,\alpha) = \pi_1 \prod_{j=1}^{D} \bar{\theta}_{jc,1}^{\mathbb{I}(x_j=1)}\left(1-\bar{\theta}_{jc,1}\right)^{\mathbb{I}(x_j=0)}$; posterior predictive mean
  - $\propto \pi_1 \prod_{j=1}^{D} \bar{\theta}_{jc,1}^{\mathbb{I}(x_j=1)}(1-\bar{\theta}_{jc,1})^{\mathbb{I}(x_j=0)}$
  - $\log\left(p(y=1|x,T)\right) = \log(\pi_1) + \sum_{j=1}^{D} \mathbb{I}\left(x_j=1\right)\log\left(\bar{\theta}_{jc,1}\right) + \sum_{j=1}^{D} \mathbb{I}\left(x_j=0\right)\log\left(1-\bar{\theta}_{jc,1}\right)$
  - π_1 is ML estimation of class 1's prior probability, simply equals to $\frac{N_1}{N}$; where *N₁* equals to number of class 1 data in training data and *N* is number of training data (3065 in this case).

- Non-spam probability (class y=0)
  - $p(y=0|x,T) = p(y=0|T)\prod_{j=1}^{D} p(x_j\,|y=0,T) = \pi_0 \prod_{j=1}^{D} p(x_j\,|y=0,\alpha)$
  - $\pi_0 \prod_{j=1}^{D} p(x_j\,|y=0,\alpha) = \pi_0 \prod_{j=1}^{D} \bar{\theta}_{jc,0}^{\mathbb{I}(x_j=1)}\left(1-\bar{\theta}_{jc,0}\right)^{\mathbb{I}(x_j=0)}$; posterior predictive mean
  - $\propto \pi_0 \prod_{j=1}^{D} \bar{\theta}_{jc,0}^{\mathbb{I}(x_j=1)}(1-\bar{\theta}_{jc,0})^{\mathbb{I}(x_j=0)}$
  - $\log\left(p(y=0|x,T)\right) = \log(\pi_0) + \sum_{j=1}^{D} \mathbb{I}\left(x_j=0\right)\log\left(\bar{\theta}_{jc,0}\right) + \sum_{j=1}^{D} \mathbb{I}\left(x_j=1\right)\log\left(1-\bar{\theta}_{jc,0}\right)$
  - π_0 is ML estimation of class 0's prior probability, equals to $\frac{N-N_1}{N}$; complement of π_1 (ie: $\pi_1 + \pi_0 = 1$

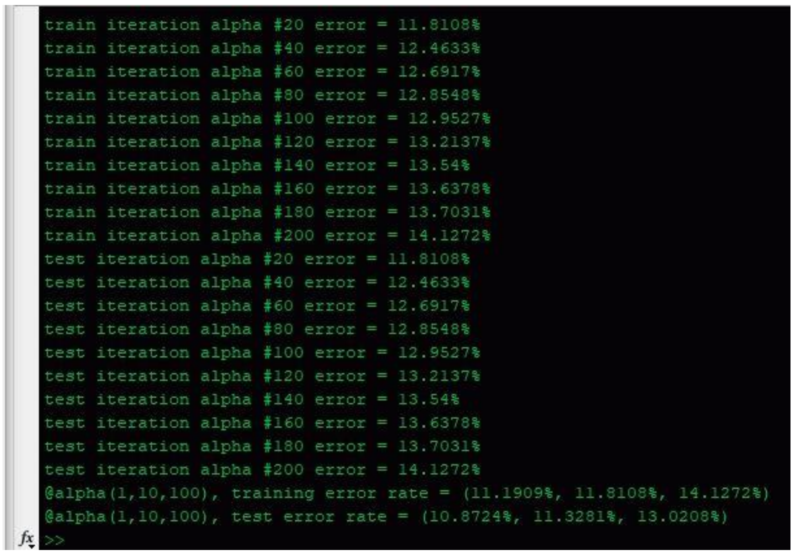
Both class probability now have log terms to be compared to classify data x into spam/non-spam. If $\log\left(p(y=0|x,T)\right) > \log\left(p(y=1|x,T)\right)$, assign label '0' (non-spam) to data x, and vice versa. The plotted result of %misclassification against alpha is shown in figure below:



Both training error and test error are increased as alpha values are being increased. Notice that there is error spike at alpha=0, due to profile of beta distribution favouring extreme values of prior feature distribution which means that prior $p\left(\theta_{jc,i}\right)$ is very high when $\theta_{jc,i} \sim 0$ (feature j-th always **<u>not</u>** equals to *i* for all data in class *c* , *i* here represents possible feature values; 1 or 0 in our binary case) or $\theta_{jc,i} \sim 1$ (feature j-th always equals to *i* for all data in class *c*), which is obviously not the case in our spamData database (training data portion) since we never see, let's say, feature 5th is always '1' for data in class spam. Therefore, error is skyrocketing when alpha = 0.

The lowest error is achieved when alpha = 1, that shows that uniform feature distribution is the closest model (alpha = 1 is better beta approximation, amongst all alpha) to the actual feature distribution in training data. To complete the list, below are the training error and test error rates for alpha = {1,10,100}:

	Alpha = 1	Alpha = 10	Alpha = 100
Training error	11.1909 %	11.8108 %	14.1272 %
Test error	10.8724 %	11.3281 %	13.0208 %



```
train iteration alpha #20 error = 11.8108%
train iteration alpha #40 error = 12.4633%
train iteration alpha #60 error = 12.6917%
train iteration alpha #80 error = 12.8548%
train iteration alpha #100 error = 12.9527%
train iteration alpha #120 error = 13.2137%
train iteration alpha #140 error = 13.54%
train iteration alpha #160 error = 13.6378%
train iteration alpha #180 error = 13.7031%
train iteration alpha #200 error = 14.1272%
test iteration alpha #20 error = 11.8108%
test iteration alpha #40 error = 12.4633%
test iteration alpha #60 error = 12.6917%
test iteration alpha #80 error = 12.8548%
test iteration alpha #100 error = 12.9527%
test iteration alpha #120 error = 13.2137%
test iteration alpha #140 error = 13.54%
test iteration alpha #160 error = 13.6378%
test iteration alpha #180 error = 13.7031%
test iteration alpha #200 error = 14.1272%
@alpha(1,10,100), training error rate = (11.1909%, 11.8108%, 14.1272%)
@alpha(1,10,100), test error rate = (10.8724%, 11.3281%, 13.0208%)
>>
```

Gaussian Naïve Bayes

Now data are pre-processed into z-normalized (zscore) and log-transform data $\log(x_{ij} + 0.1)$ and are to be fitted into Gaussian Naïve Bayes Model. Since now the data are 'relatively' continuous, not like binarized case in Question 1, there should be mean and variance of each feature in each class and in each data type. To find these means and variances that maximize the likelihood of feature values to be the same in the data above, I use ML estimation of a univariate gaussian distribution :

$$(\hat{\mu}_j, \hat{\sigma}_j^2) = \underset{\mu,\sigma^2}{\operatorname{argmax}} \sum_{n=1}^{N} \left(-\frac{(x_n - \mu_j)^2}{2\sigma_j^2} - 0.5\log(2\pi\sigma_j^2) \right)$$

Which ended up in the forms below after some differentiation to find maxima condition:

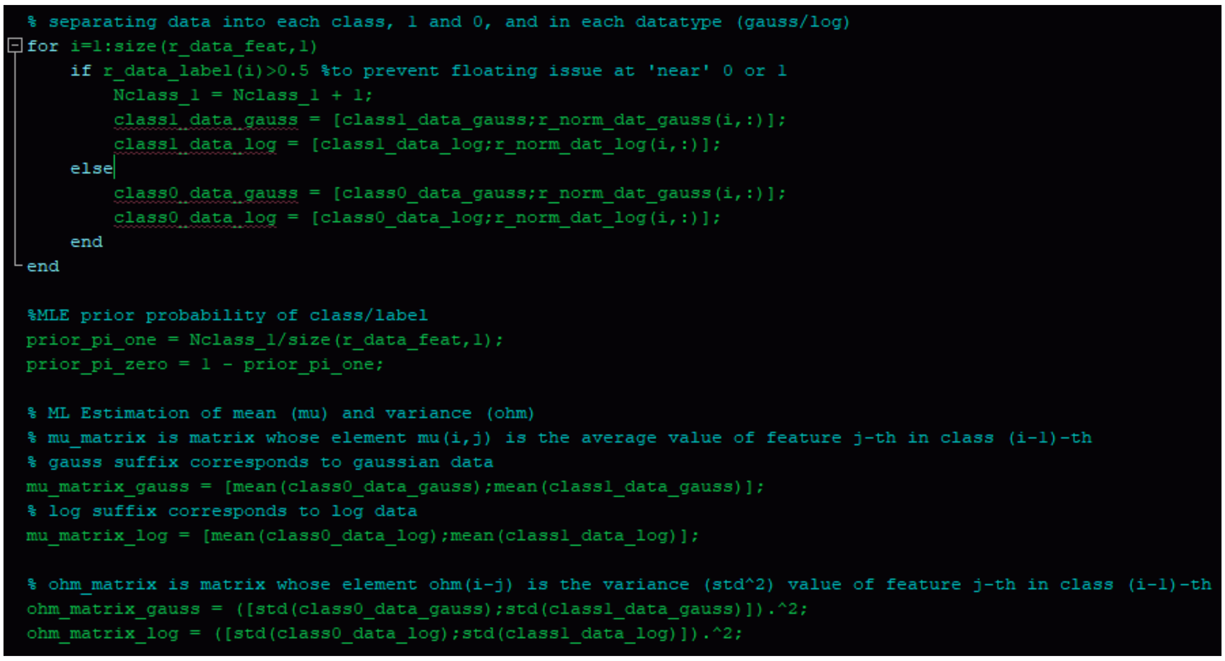
$$\frac{\partial}{\partial \mu} \sum_{n=1}^{N} \left(-\frac{(x_n - \mu_j)^2}{2\sigma_j^2} - 0.5\log(2\pi\sigma_j^2) \right) = \sum_{n=1}^{N} \frac{(x_n - \hat{\mu}_j)}{\sigma_j^2} = 0 \xrightarrow{yields} \hat{\mu}_j = \frac{1}{N} \sum_{n=1}^{N} x_n$$

And the ML estimation of variance is given by :

$$\frac{\partial}{\partial \sigma_j^2} \sum_{n=1}^{N} \left(-\frac{(x_n - \hat{\mu}_j)^2}{2\sigma_j^2} - 0.5\log(2\pi\sigma_j^2) \right) = \sum_{n=1}^{N} \left(\frac{(x_n - \hat{\mu}_j)^2}{2\hat{\sigma}_j^4} - \frac{2\pi}{4\pi\hat{\sigma}_j^2} \right) = 0$$

$$\sum_{n=1}^{N} \left(\frac{(x_n - \hat{\mu}_j)^2}{\hat{\sigma}_j^2} - 1 \right) = 0 \to \frac{1}{\hat{\sigma}_j^2} \sum_{n=1}^{N} (x_n - \hat{\mu}_j)^2 - N = 0 \xrightarrow{yields} \frac{1}{N} \sum_{n=1}^{N} (x_n - \hat{\mu}_j)^2 = \hat{\sigma}_j^2$$

With these equations, I can find the values of means and variances of each feature in each data, yielding matrix $\mu = \{\hat{\mu}_{cj}\}_{c=0:1;j=1:57}$ and matrix $\sigma^2 = \{\hat{\sigma}^2_{cj}\}_{c=0:1;j=1:57}$ where j is feature number and c is class number. Since I have 57 features and two classes, these two matrices will have size 2-by-57, which in the code is obtained in expression below :



```
% separating data into each class, 1 and 0, and in each datatype (gauss/log)
for i=1:size(r_data_feat,1)
    if r_data_label(i)>0.5 %to prevent floating issue at 'near' 0 or 1
        Nclass_1 = Nclass_1 + 1;
        class1_data_gauss = [class1_data_gauss;r_norm_dat_gauss(i,:)];
        class1_data_log = [class1_data_log;r_norm_dat_log(i,:)];
    else
        class0_data_gauss = [class0_data_gauss;r_norm_dat_gauss(i,:)];
        class0_data_log = [class0_data_log;r_norm_dat_log(i,:)];
    end
end

%MLE prior probability of class/label
prior_pi_one = Nclass_1/size(r_data_feat,1);
prior_pi_zero = 1 - prior_pi_one;

% ML Estimation of mean (mu) and variance (ohm)
% mu_matrix is matrix whose element mu(i,j) is the average value of feature j-th in class (i-1)-th
% gauss suffix corresponds to gaussian data
mu_matrix_gauss = [mean(class0_data_gauss);mean(class1_data_gauss)];
% log suffix corresponds to log data
mu_matrix_log = [mean(class0_data_log);mean(class1_data_log)];

% ohm_matrix is matrix whose element ohm(i-j) is the variance (std^2) value of feature j-th in class (i-1)-th
ohm_matrix_gauss = ([std(class0_data_gauss);std(class1_data_gauss)]).^2;
ohm_matrix_log = ([std(class0_data_log);std(class1_data_log)]).^2;
```

Prior class probability is similarly estimated using ML like in the previous question (i.e., $\frac{N_1}{N}$ for class 1), and the posterior class probability given data is also in similar form except the forms inside sum operator:

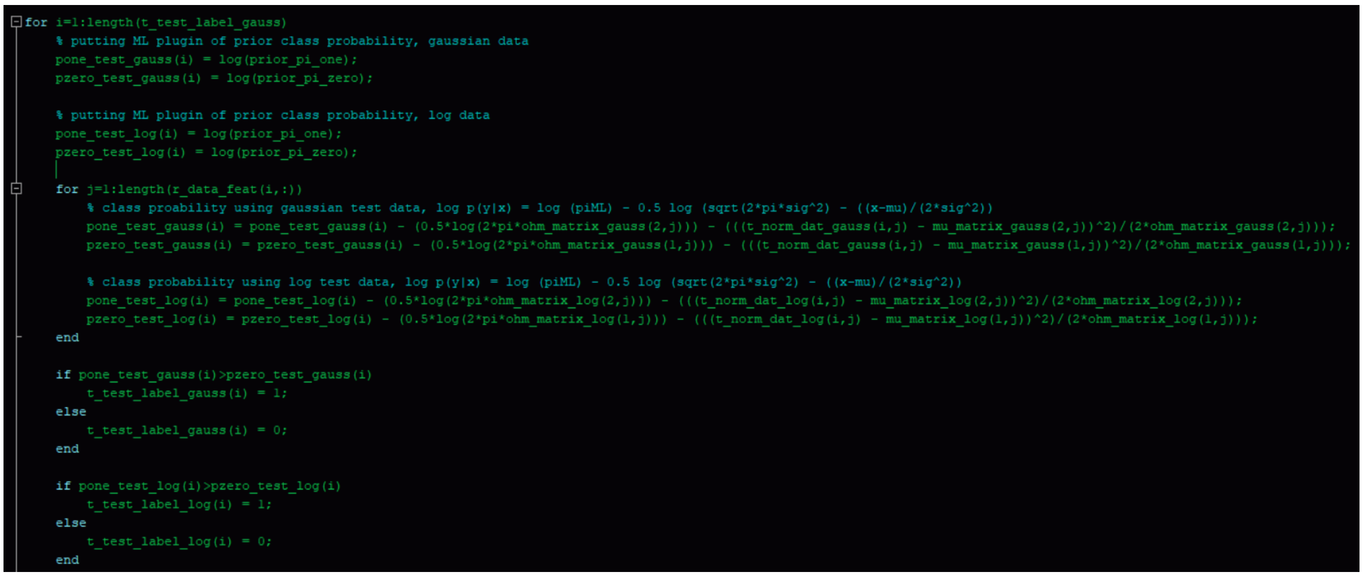
- Non-spam log probability

$$\log\big(p(y=0|x,T)\big) = \log(\pi_0) - 0.5\sum_{j=1}^{D}\log\left(2\pi\hat{\sigma}^2_{0j}\right) - \sum_{j=1}^{D}\frac{\left(x_n - \hat{\mu}_{0j}\right)^2}{2\hat{\sigma}^2_{0j}}$$

- Spam log probability

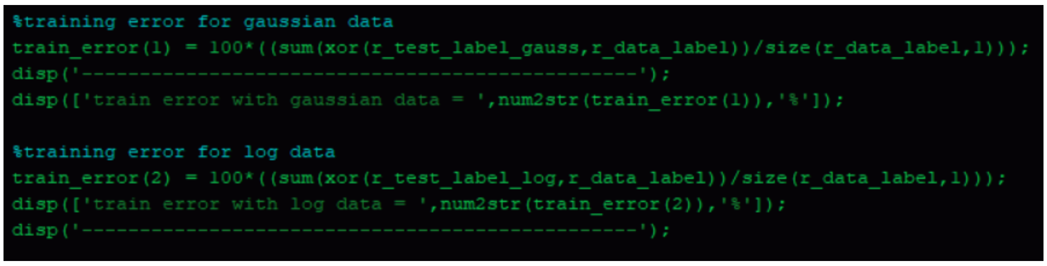
$$\log\big(p(y=1|x,T)\big) = \log(\pi_1) - 0.5\sum_{j=1}^{D}\log\left(2\pi\hat{\sigma}^2_{1j}\right) - \sum_{j=1}^{D}\frac{\left(x_n - \hat{\mu}_{1j}\right)^2}{2\hat{\sigma}^2_{1j}}$$

And compare both probability to assign label to data x. Code for above class posterior probability computation is shown below:

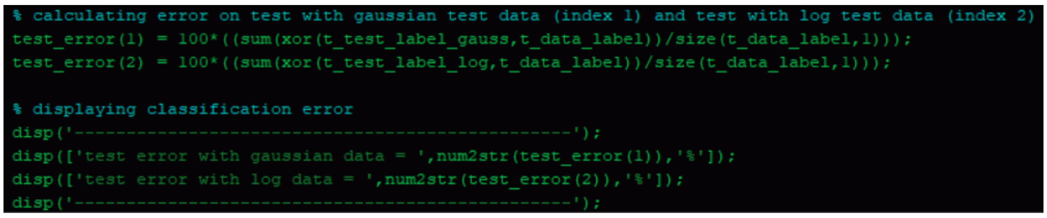


```
for i=1:length(t_test_label_gauss)
    % putting ML plugin of prior class probability, gaussian data
    pone_test_gauss(i) = log(prior_pi_one);
    pzero_test_gauss(i) = log(prior_pi_zero);

    % putting ML plugin of prior class probability, log data
    pone_test_log(i) = log(prior_pi_one);
    pzero_test_log(i) = log(prior_pi_zero);

    for j=1:length(r_data_feat(i,:))
        % class proability using gaussian test data, log p(y|x) = log (piML) - 0.5 log (sqrt(2*pi*sig^2) - ((x-mu)/(2*sig^2))
        pone_test_gauss(i) = pone_test_gauss(i) - (0.5*log(2*pi*ohm_matrix_gauss(2,j))) - (((t_norm_dat_gauss(i,j) - mu_matrix_gauss(2,j))^2)/(2*ohm_matrix_gauss(2,j)));
        pzero_test_gauss(i) = pzero_test_gauss(i) - (0.5*log(2*pi*ohm_matrix_gauss(1,j))) - (((t_norm_dat_gauss(i,j) - mu_matrix_gauss(1,j))^2)/(2*ohm_matrix_gauss(1,j)));

        % class probability using log test data, log p(y|x) = log (piML) - 0.5 log (sqrt(2*pi*sig^2) - ((x-mu)/(2*sig^2))
        pone_test_log(i) = pone_test_log(i) - (0.5*log(2*pi*ohm_matrix_log(2,j))) - (((t_norm_dat_log(i,j) - mu_matrix_log(2,j))^2)/(2*ohm_matrix_log(2,j)));
        pzero_test_log(i) = pzero_test_log(i) - (0.5*log(2*pi*ohm_matrix_log(1,j))) - (((t_norm_dat_log(i,j) - mu_matrix_log(1,j))^2)/(2*ohm_matrix_log(1,j)));
    end

    if pone_test_gauss(i)>pzero_test_gauss(i)
        t_test_label_gauss(i) = 1;
    else
        t_test_label_gauss(i) = 0;
    end

    if pone_test_log(i)>pzero_test_log(i)
        t_test_label_log(i) = 1;
    else
        t_test_label_log(i) = 0;
    end
```

The training error and test error for both datatype are then calculated based on mismatching between the ‘new’ label against original label of both training and test data. To get easy computation, the label vectors are turned into column vector and perform sum(xor(newlabel,original_label)) as shown below:

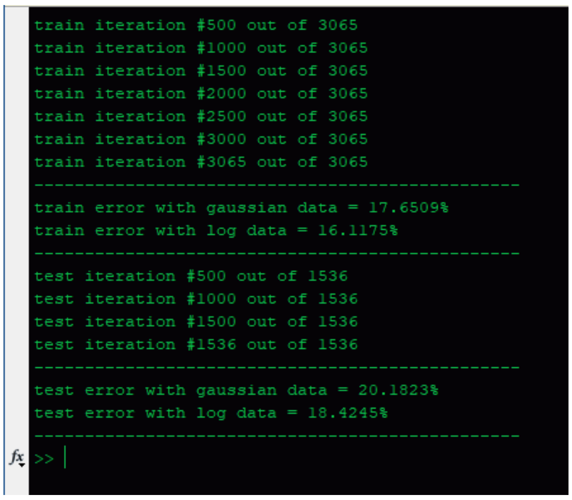


```
%training error for gaussian data
train_error(1) = 100*((sum(xor(r_test_label_gauss,r_data_label))/size(r_data_label,1)));
disp('------------------------------------------------');
disp(['train error with gaussian data = ',num2str(train_error(1)),'%']);

%training error for log data
train_error(2) = 100*((sum(xor(r_test_label_log,r_data_label))/size(r_data_label,1)));
disp(['train error with log data = ',num2str(train_error(2)),'%']);
disp('------------------------------------------------');
```



```
% calculating error on test with gaussian test data (index 1) and test with log test data (index 2)
test_error(1) = 100*((sum(xor(t_test_label_gauss,t_data_label))/size(t_data_label,1)));
test_error(2) = 100*((sum(xor(t_test_label_log,t_data_label))/size(t_data_label,1)));

% displaying classification error
disp('------------------------------------------------');
disp(['test error with gaussian data = ',num2str(test_error(1)),'%']);
disp(['test error with log data = ',num2str(test_error(2)),'%']);
disp('------------------------------------------------');
```

The result of fitting data from both datatypes into gaussian naïve bayes model is given below :



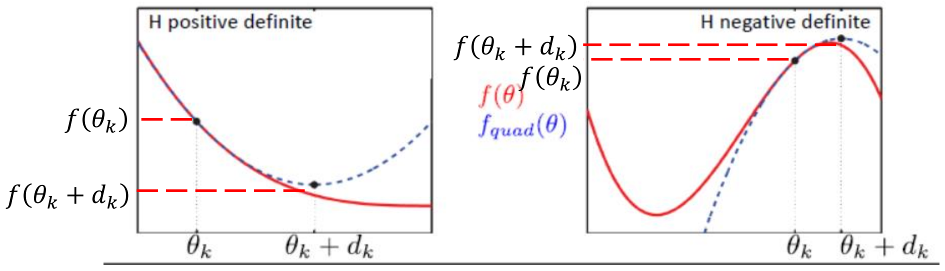
```
train iteration #500 out of 3065
train iteration #1000 out of 3065
train iteration #1500 out of 3065
train iteration #2000 out of 3065
train iteration #2500 out of 3065
train iteration #3000 out of 3065
train iteration #3065 out of 3065
------------------------------------------------
train error with gaussian data = 17.6509%
train error with log data = 16.1175%
------------------------------------------------
test iteration #500 out of 1536
test iteration #1000 out of 1536
test iteration #1500 out of 1536
test iteration #1536 out of 1536
------------------------------------------------
test error with gaussian data = 20.1823%
test error with log data = 18.4245%
------------------------------------------------
>>
```

	Z-normalized	Log
Training error	17.6509 %	16.1175 %
Test error	20.1823 %	18.4245 %

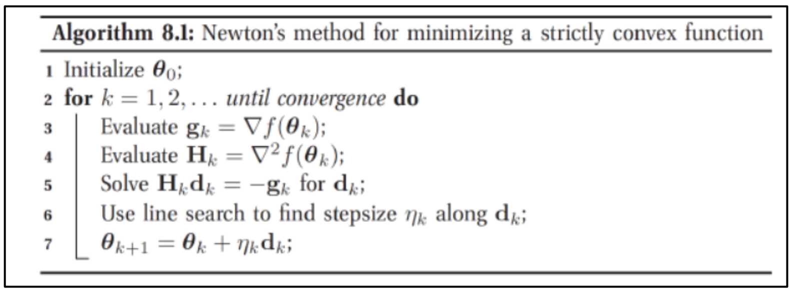
Log pre-processing performed better in this scenario since log is naturally maintaining the relative value of data (monotonic), in the sense that it likely doesn’t ‘alter’ the data label during re-labelling process (i.e. picking new label after comparing class posterior probability obtained from plug-in testing) since log pre-processing relatively maintain distribution of data (i.e. the absolute values might be changed but it is relatively not changed between one data compared to other data). Z-normalized data forced the data distribution to follow normal distribution which ‘reshape’ feature value’s distribution, and will potentially affect/alter its class label since feature values are the determining factors as given in the spamData (i.e. spamData only has feature values and label, I need to guess the ‘new’ label based on pattern in the feature values). If feature value’s distribution is re-shaped, then the class label will likely be changed, and causing more errors.

<u>Newton's Logistic Regression</u>

The task in this exercise is to perform log regression to each version of data (z-normalized, log-transformed and binarized). To add, the regression must include l_2 regularization with parameter value $\lambda = \{1,2,3,..,9,10,15,20,..95,100\}$. In matlab, this lambda will be a vector [(1:1:9) (10:5:100)]. The algorithm is coded in such a way that regularization is not applied onto bias term. Initial weight vector w is set at large value and its dimension is 58-by-1, the reason of one additional element is to add bias weight w_o on top of the number of weights for each dimension/feature (57 features). With the weight vector set in that way, I should add a row vector of 1 in the matrix that concatenate all training data or test data. By using Taylor expansion to find changes of θ_k, which is d_k, I can find d_k that minimize the quadratic approximation of Negative Log Likelihood function at θ_k. However, this Newton's method requires hessian matrix *H* to be positive, otherwise the $d_k$will lead to higher NLL value and may move the iteration away from weight convergence. Graphically, it is explained below:



When *H* is positive definite matrix, approximated quadratic function (blue dotted line) gives d_k that lead to lower NLL value in the red-line function, i.e. $f(\theta_k + d_k) < f(\theta_k)$. When it is in the opposite case, *H* doesn't guarantee that the next value $f(\theta_k + d_k)$ is lower than current value $f(\theta_k)$. In this problem, *H* is more likely positive definite matrix otherwise Question 3 is not do-able using Newton's Method for Logistic Regression as it has asked. So the MATLAB algorithm used in this answer will not check matrix *H* 's positive definitive condition. The algorithm is pretty much described in the pseudo code below :



```
Algorithm 8.1: Newton's method for minimizing a strictly convex function
1 Initialize $\boldsymbol{\theta}_0$;
2 for $k = 1, 2, \ldots$ until convergence do
3     Evaluate $\mathbf{g}_k = \nabla f(\boldsymbol{\theta}_k)$;
4     Evaluate $\mathbf{H}_k = \nabla^2 f(\boldsymbol{\theta}_k)$;
5     Solve $\mathbf{H}_k \mathbf{d}_k = -\mathbf{g}_k$ for $\mathbf{d}_k$;
6     Use line search to find stepsize $\eta_k$ along $\mathbf{d}_k$;
7     $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \eta_k \mathbf{d}_k$;
```

Except that the value of learning rate/stepsize is not determined by line search, instead it is set at fixed constant ($\eta = 1$). Hence, the last term is just simply adding d_k to θ_k in order to get the next argument θ_{k+1}. Incorporating the regularization constant in the gradient and hessian term, the new gradient and hessian term being used are mathematically illustrated below:

$$g_{reg}(w) = g(w) + \lambda w \rightarrow g_{reg}(w) = X^T(\mu - y) + \lambda w \xrightarrow{to\ not\ affect\ bias\ term} g_{reg}(w)$$
$$= \begin{pmatrix} 1 \\ X^T \end{pmatrix} (\mu - y) + (0 \quad \lambda) w$$

The term 1 in $\begin{pmatrix} 1 \\ X^T \end{pmatrix}$ is a row vector whose length matches the column size of X^T (which is number of data, 3065). And the term 0 in $(0 \quad \lambda)$ is a scalar that accompanies vector λ (1-by-57) which is intended to avoid multiplying bias weight in the weight vector w. As for the Hessian matrix, the form is changed to below to avoid applying regularization constant to term that is related to bias weight.

$$H_{reg}(w) = H(w) + \lambda I \rightarrow H_{reg}(w) = X^T S X + \lambda I \xrightarrow{to\ not\ affect\ bias\ term} H_{reg}(w)$$
$$= \begin{pmatrix} 1 \\ X^T \end{pmatrix} S \begin{pmatrix} 1 \\ X^T \end{pmatrix}^T + \lambda \begin{pmatrix} 0 & 0 \\ 0 & I_{57\times 57} \end{pmatrix}$$

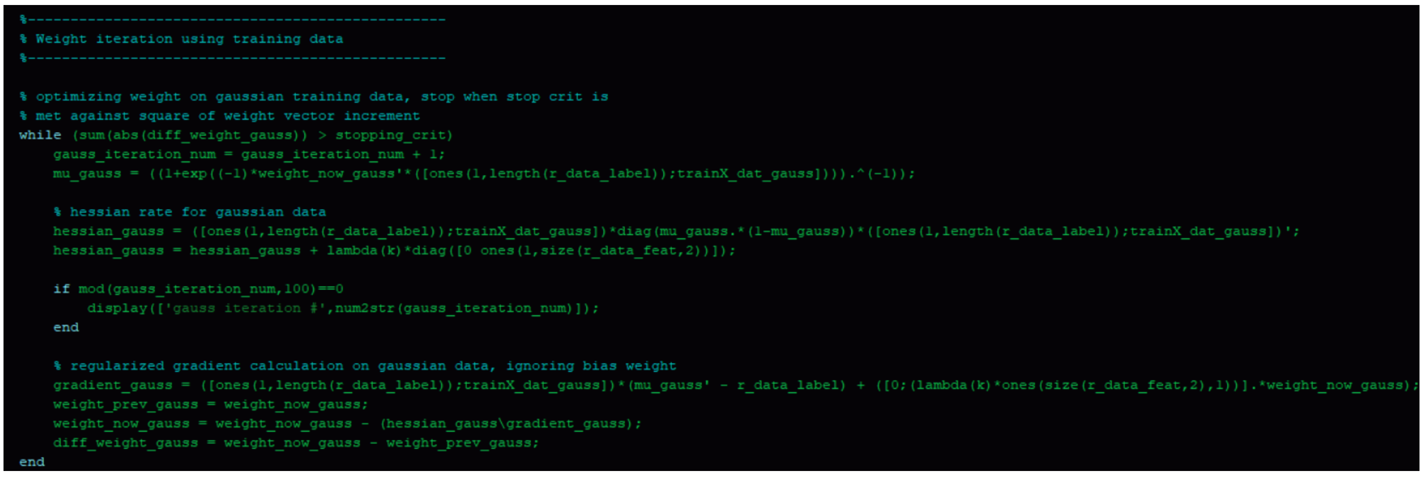
All these gradient and hessian terms are used to calculate the next weight as shown below:

$$w_{k+1} = w_k - H_{reg}(w_k)^{-1} g_{reg}(w_k)$$

The weight update above is continuously being iterated until convergence condition is met. In this exercise, the stopping criteria of 'while' update loop is :

$$|w_{k+1} - w_k| < 1 \times 10^{-5}$$

The MATLAB code is shown below:

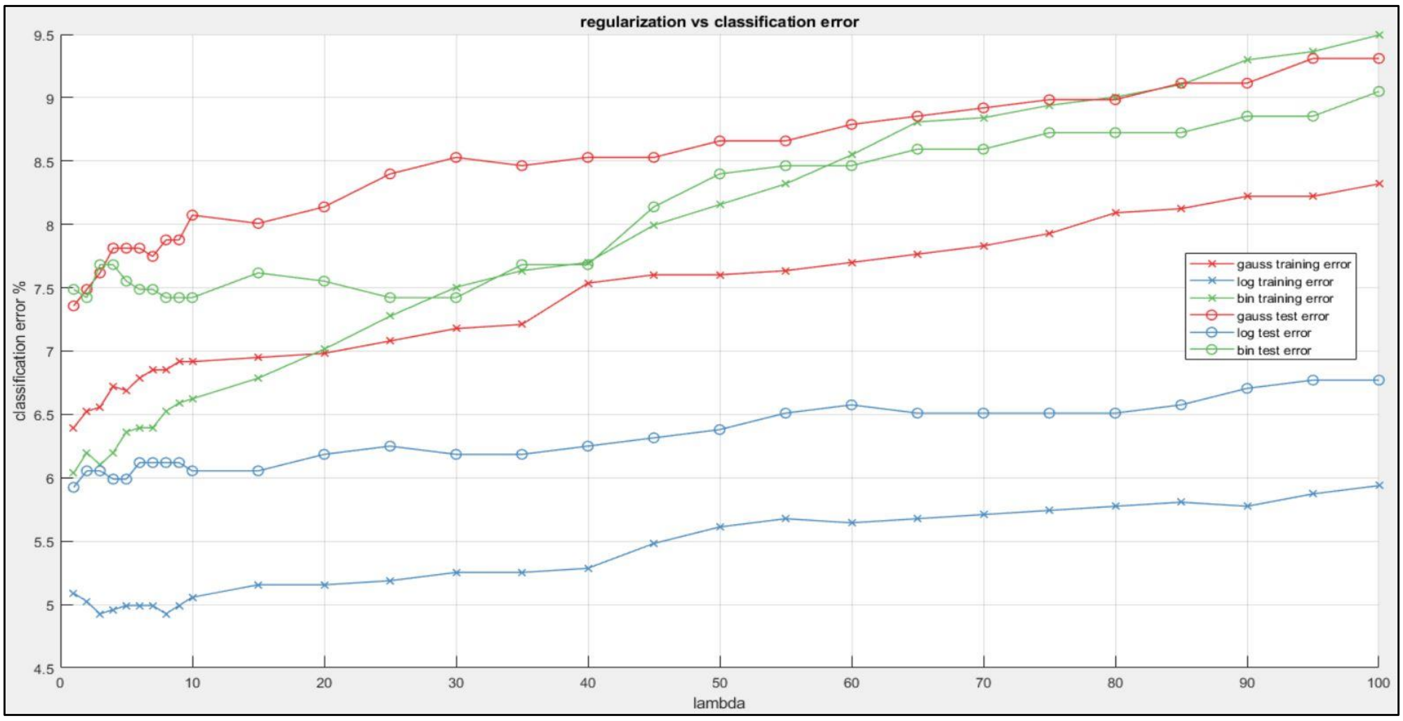


```
%-----------------------------------------------
% Weight iteration using training data
%-----------------------------------------------

% optimizing weight on gaussian training data, stop when stop crit is
% met against square of weight vector increment
while (sum(abs(diff_weight_gauss)) > stopping_crit)
    gauss_iteration_num = gauss_iteration_num + 1;
    mu_gauss = ((1+exp((-1)*weight_now_gauss'*([ones(1,length(r_data_label));trainX_dat_gauss]))).^(-1));

    % hessian rate for gaussian data
    hessian_gauss = ([ones(1,length(r_data_label));trainX_dat_gauss])*diag(mu_gauss.*(1-mu_gauss))*([ones(1,length(r_data_label));trainX_dat_gauss])';
    hessian_gauss = hessian_gauss + lambda(k)*diag([0 ones(1,size(r_data_feat,2))]);

    if mod(gauss_iteration_num,100)==0
        display(['gauss iteration #',num2str(gauss_iteration_num)]);
    end

    % regularized gradient calculation on gaussian data, ignoring bias weight
    gradient_gauss = ([ones(1,length(r_data_label));trainX_dat_gauss])*(mu_gauss' - r_data_label) + ([0;(lambda(k)*ones(size(r_data_feat,2),1))].*weight_now_gauss);
    weight_prev_gauss = weight_now_gauss;
    weight_now_gauss = weight_now_gauss - (hessian_gauss\gradient_gauss);
    diff_weight_gauss = weight_now_gauss - weight_prev_gauss;
end
```

Where stopping_crit = 1×10^{-5} to break while loop and move on. The converged weight w_f is then used to label the data x by comparing class posterior probability $p(y=1|x) = \frac{1}{1+e^{-w_f{}^T x}}$ and $p(y=0|x) = \frac{1}{1+e^{w_f{}^T x}}$. If equality between the two is met ($p(y=1|x) == p(y=0|x)$, I can arbitrarily assign any label (1 or 0), or I can call random function to assign the label for 'fairer/un-biased' labelling. The result of MATLAB code executed to answer question three is tabularized below:

	Training error			Test error		
	λ=1	λ=10	λ=100	λ=1	λ=10	λ=100
Z-normalized	6.3948 %	6.9168 %	8.3197 %	7.3568 %	8.0729 %	9.0399 %
Log-Transform	5.0897 %	5.0571 %	5.938 %	5.9245 %	6.0547 %	6.7708 %
Binarized	6.0359 %	6.232 %	9.4943 %	7.487 %	7.4219 %	9.0495 %

And to observe the overall classification performance over defined regularization value $\lambda = \{1,2,3,..,9,10,15,20,..95,100\}$, the plot is presented below:



Generally, looking at the trend of the plots, both training and test error increase as regularization constant grows. The growing of regularization constant gives more restriction on the absolute weight values to grow uncontrollably. When absolute weight values are significantly large, the steepness of sigmoid function is closer to that straight line at $x_{boundary}$, which leading to overfitting. Therefore, restricting absolute weight values to grow to large values by putting regularization constant, at some point might lead to better generalization. But if regularization constant is set at even higher values, the sigmoid's gradient will not be 'steep' enough to classify the data properly.

In terms of data pre-processing, log error, again, performed better in both training data and test data. As evaluated earlier, log-transform doesn't change the data values relative to other data. In other words, log-transform still keeps the relative 'distances' of all data due to its monotonicity. Z-normalized and binary are not monotonic transformation, thus they possibly change the relative values of the data.

One more fashion observed in the plots, training error are relatively lower than test error for Z-normalized data and log-transformed data. This could be due to the poor generalization after applying regularization which is intuitively, by looking at the graph, not suitable for Z-normalized and log-transformed data. On the other hand, binarized data has slightly 'better' generalization from training data to the test data evidenced by close plot between training error and test error at some lambda range ($\lambda \sim 30 - 60$). However, if λ continues to grow, binarized data has even higher error rates.

KNN classifier

In this section, the task is to re-label training data and test data using k-NN algorithm. The neighbour size used to perform re-labelling is varied from the range $K = \{1,2,3,..,9,10,15,20,\ldots 95,100\}$, which requires iteration on each K. The neighbouring is determined by:

- The Euclidean distance for Z-normalized and log-transformed data:

  $d_{ij} = \sqrt{(x_i - x_j)^T (x_i - x_j)}$; x_i, x_j are assumed as column vectors

- Hamming distance for binarized data:

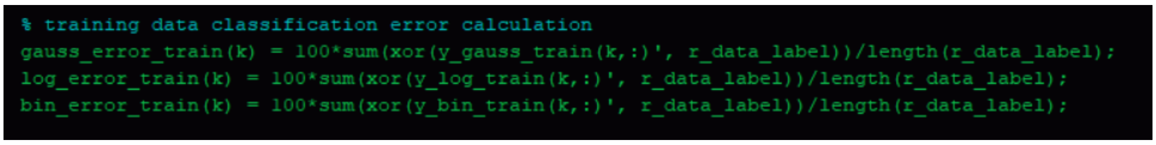
  $d_{ij} = sum\left(XOR(x_i, x_j)\right)$; x_i, x_j are also assumed as column binary vectors

  This hamming distance is basically calculating the number of different feature values between the two binary data vectors.

And the pseudocode for this k-NN classifier is illustrated below:

```
1.  Calculate distance $r_{ij}$ from one training data $i$ to other training data $j$ .
2.  Repeat step 1 to every single data in training data and create fetch-able matrix from it.
    $i$ =1:3065 and $j$ = 1:3065
3.  Set $r_{ij} = 1 \times 10^3$ if $i = j$ in order to get rid of 'self-distance' which can cause sorting issue later.
4.  Calculate distance $t_{ij}$ between data in test data $i$ and data $j$ in training data.
5.  Repeat step 4 to every single data in test data and create fetch-able matrix from it.
    $i$ =1:3065 and $j$ = 1:3065.
6.  For each k in vector K = {1,2,3,…9,10,15,20,….,95,100}
7.      For each data $i$ in training data
8.          Sort distance vector $r_{i,:}$
9.          Pick the labels of nearest k neighbours based on sorted distance.
10.         Find the majority label $c$ from that k neighbours.
11.         Assign label $c$ to data $i$ .
12.     End;
13.     For each data $m$ in test data.
14.         Sort distance vector $t_{m,:}$
15.         Pick the labels of nearest k neighbours based on sorted distance.
16.         Find the majority label $c$ from that k neighbours.
17.         Assign label $c$ to data $i$ .
18.     End;
19. End;
```

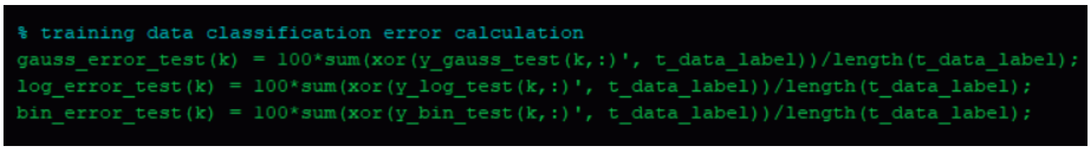
After getting the new label of training data and test data, I then can compute the mislabelling by comparing the new label and original label. The easier way to compute mislabelling is using sum and XOR function from MATLAB, as shown in the code below:



```
% training data classification error calculation
gauss_error_train(k) = 100*sum(xor(y_gauss_train(k,:)', r_data_label))/length(r_data_label);
log_error_train(k) = 100*sum(xor(y_log_train(k,:)', r_data_label))/length(r_data_label);
bin_error_train(k) = 100*sum(xor(y_bin_train(k,:)', r_data_label))/length(r_data_label);
```

In the picture above, ***y_gauss_train(k , :)*** is a vector that contains the new label for z-normalized training data for **k** nearest neighbours. When I XOR it with original training label ***r_data_label***, I will get a vector that contains 3065 elements of either 0 or 1, '1' for misclassified and '0' for correct classification. I can simply sum up this XOR vector and divide by 3065 (size of training data, **length(*r_data_label*)**) and multiply by 100 for percentage representation.

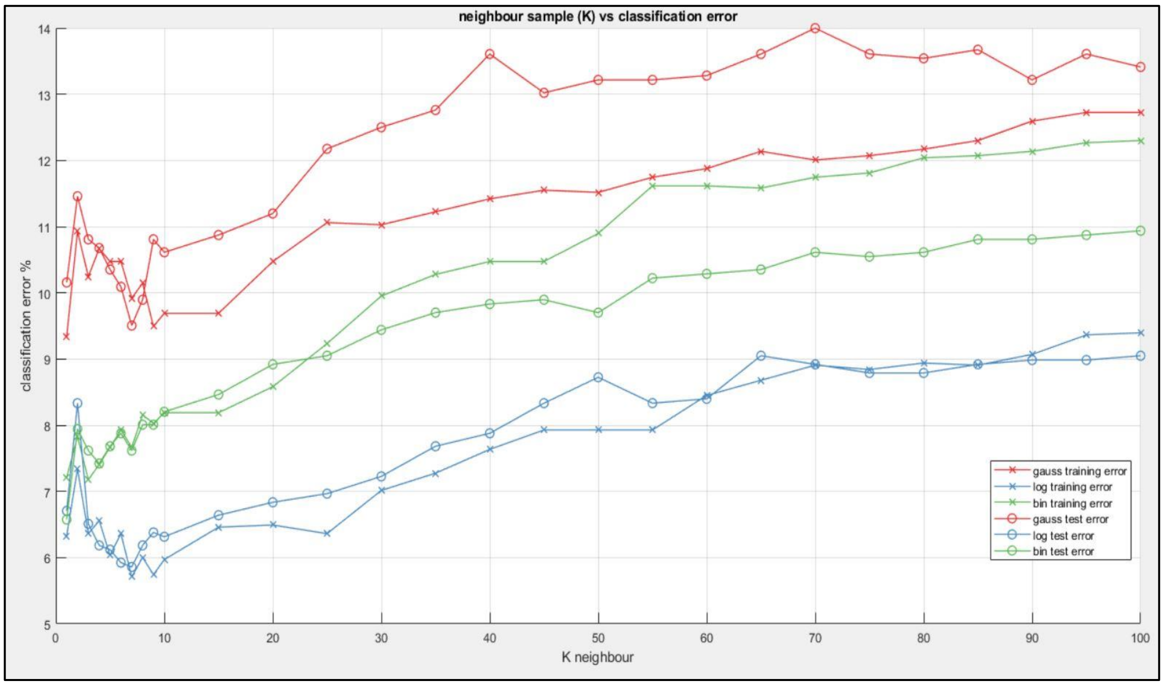
The same thing can be done for test data, as shown below:



```
% training data classification error calculation
gauss_error_test(k) = 100*sum(xor(y_gauss_test(k,:)', t_data_label))/length(t_data_label);
log_error_test(k) = 100*sum(xor(y_log_test(k,:)', t_data_label))/length(t_data_label);
bin_error_test(k) = 100*sum(xor(y_bin_test(k,:)', t_data_label))/length(t_data_label);
```

The suffices are now changed to 'test' to indicate the computation is performed on test data. And instead of ***r_data_label***, I put ***t_data_label*** in place so the error is relative to size of test data, which how it should be calculated, based on 1536 test data.

The result of the code execution in MATLAB is plotted in the figure below:



The classification error increases as ***K*** increases regardless which pre-processing is utilized. Training error is not 0% when ***K=1*** since the nearest neighbour doesn't always the same label as the subject data point. That means training data are scattered around almost in alternate fashion between class spam ('1') and non-spam('0'). This can be verified with result when ***K=2*** , where the error suddenly picks a sudden surge-up, especially in continuous data (z-normalized data and log-transformed data), which indicates a 50-50 condition is frequently met where the two neighbours have different label. That condition leads to arbitrary label assignment on the data that encounters 50-50 condition.

In terms of pre-processing, log-transformed data outperformed other pre-processing method as what we have seen in the previous questions, it is due to its monotonic transformation that doesn't change the distribution of original data.

To summarize the classification error, the table below presents the error in 10th multiplier of ***K***.

	Training error			Test error		
	K=1	K=10	K=100	K=1	K=10	K=100
Z-normalized	9.3312 %	9.7879 %	12.6591 %	10.2214 %	10.5469 %	13.4115 %
Log-Transform	6.2969 %	6.3948 %	9.3312 %	6.7708 %	6.2500 %	9.0495 %
Binarized	7.5693 %	8.2545 %	12.3654 %	6.7057 %	8.0729 %	10.9375 %

<u>End of Report</u>