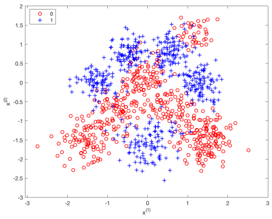
Exercise: a) Load data and visualise it using gscatter:



```
load('classification.mat');
gscatter(X(:,1), X(:,2), y, 'rb','o+');
xlabel('x^{(1)}'), ylabel('x^{(2)}');
```

Exercise: b) Use holdout function to split data set into training set and test set with some `trainRatio` while keeping 0/1s ratio of original data set in both. `trainRatio = 0.6` should ensure sufficient accuracy later on the test set.

```
function [X_train, X_test, y_train, y_test] = holdout(X, y, trainRatio)
N = size(X, 1);
y_0 = find(y == 0);      %Array of sample entries x_i with y = 0
T_0 = datasample(y_0, floor(trainRatio*size(y_0,1)),'Replace',false);
y_1 = find(y);           %Array of sample entries x_i with y = 1
T_1 = datasample(y_1, floor(trainRatio*size(y_0,1)),'Replace',false);

ind_train = [T_0;T_1]; ind_test = setdiff(1:N, ind_train); %indices
X_train = X(ind_train,:); y_train = y(ind_train);
X_test = X(ind_test,:); y_test = y(ind_test);
end
```

Exercise: c) Consider a Logistic Classification model for extended inputs $\tilde{x}_n = (1, x_n)^T$ for $n = 1, ..., N$ with parameter β:

$$P(y_n = 1 \mid \tilde{x}_n) = \frac{1}{1 + exp(-\beta^T \tilde{x}_n)} := \sigma(\beta^T \tilde{x}_n)$$

Loglikelihood: $\mathcal{L}(\beta) = \sum_{n=1}^{N} y_n log(\sigma(\beta^T \tilde{x}_n)) + (1 - y_n) log(1 - \sigma(\beta^T \tilde{x}_n))$

Gradient of loglikelihood using derivative identity of σ:

$$\frac{\partial \mathcal{L}(\beta)}{\partial \beta} = \sum_{n=1}^{N} y_n \frac{(1 - \sigma(\beta^T \tilde{x}_n))\sigma(\beta^T \tilde{x}_n)}{\sigma(\beta^T \tilde{x}_n)} \tilde{x}_n - (1 - y_n) \frac{(1 - \sigma(\beta^T \tilde{x}_n))\sigma(\beta^T \tilde{x}_n)}{1 - \sigma(\beta^T \tilde{x}_n)} \tilde{x}_n = \sum_{n=1}^{N} [y_n - \sigma(\beta^T \tilde{x}_n)] \tilde{x}_n$$

We compute the loglikelihood for β and its gradient given any extended data set `X_ext` and labels `y` with:

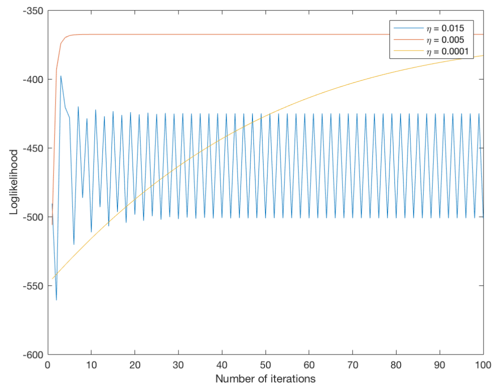
```
function [loglike, grad] = log_likelihood_grad(X_ext, y, beta)
N = size(X_ext,1);
%My function for Logistic classicication model P(y=1|x_ext) = sigma(beta'x_ext)
sigma = probs_logreg(X_ext, beta);
%Initialise output variables
loglike = 0; grad=0;
%Term by term sum
for n=1:N
    sum_term = y(n)*log(sigma(n)) + (1 - y(n))*log(1 - sigma(n));
    loglike = loglike + sum_term;
    grad_term = (y(n) - sigma(n))*X_ext(n,:)';
    grad = grad + grad_term;
end
end
```

Exercise: d) The script below shows the implementation of gradient ascent (with some learning rate η) for learning parameters β that maximise the loglikelihood on the training set:

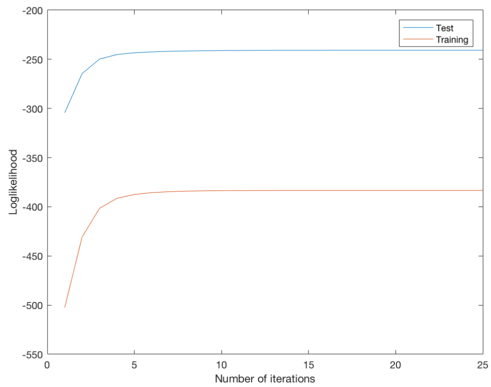
```
function [beta, loglike_track] = train_logreg(X_ext,y, learning_rate)
N = size(X_ext,1); D = size(X_ext,2);
loglike_track = []; i = 1;
%Initialise Beta  and grad from Multivariate Gaussian Distribution
grad = mvnrnd(zeros(D,1), eye(D ))'; beta = mvnrnd(zeros(1, D), eye(D))';

while norm(grad) > 0.01 %Expecting grad ~ 0 around a local max
    %Call loglikelihood and its gradient
    [loglike, grad] = log_likelihood_grad(X_train_ext, y_train, beta);
    %Updating beta and number of iterations
    beta = beta + learning_rate * grad;
    %Keeping track of the loglikelihood fot later plotting
    loglike_track = [loglike_track, loglike];
    %Convergence control
        if i > 100
            disp('Convergence not achieved after 100 iterations')
            break
            continue
        end
    i = i + 1;
end
end
```

The learning rate η has to be small enough, so that the loglikelihood does not oscillate avoiding convergence; and large enough, so that the convergence does not take an infinite amount of time. After some experimentation (shown in the figure below) `learning_rate = 0.005` seems to be the best candidate in terms of number of iterations to convergence.



(a) Gradient ascent with different η values

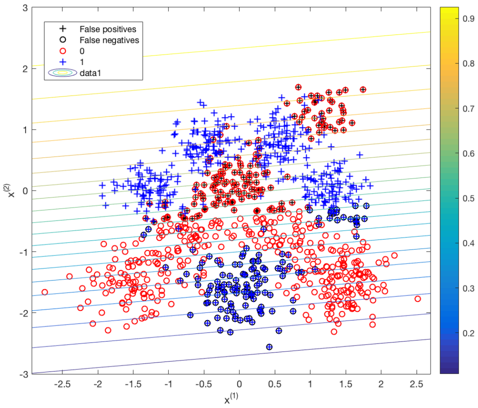


(b) Loglikelihoods for training and test sets

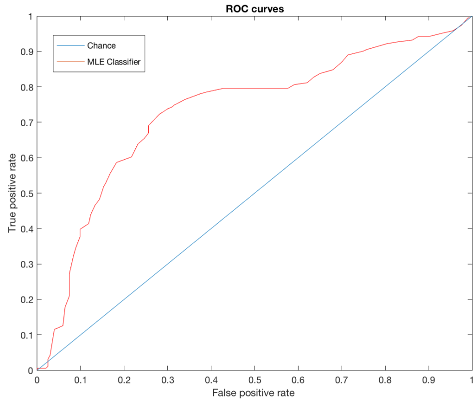
Alternatively, we can choose a *adaptative* learning rate that adjust its size depending on the grad size and iteration number, but it is beyond the scope of this coursework.

Exercise: e) Train the Logistic Classification model on training set `X_train_ext` via:
`train_logreg(X_train_ext,y_train, 0.005)` it returns: `beta = [0.3178; -0.1467; 0.8206]`

The figure below displays the probabilty contour $P(y = 1 \mid \tilde{x}, \hat{\beta})$ (with $\hat{\beta}$ as our previous MLE of β) and the original data points. Marked in black the prediction mistakes (false positives and negatives) if a Bayes classifier was used (hard decisions).



(a) Probabilty contour using MLE of β



(b) ROC curves of MLE vs chance classifier

Exercise: f) We can compute the final training and test likelihoods per data point obtaining

`loglike_train_pp = -0.6404`, `loglike_test_pp = -238.9777`

Using the script on `X_test_ext` with threshold `tau = 0.5` returns (hard) predictions `y_pred`$\in \{0, 1\}$ and the corresponding confusion matrix

```
function [y_pred, confusion_mat] = predicts(y, X_ext, beta, tau)
    N = size(X_ext,1);
    %Predictions based on probabilities P(y = 1 | x, beta)
    probs = probs_logreg(X_ext, beta);      %N-column vector with probailities
    y_pred = probs > tau*ones(N,1);         %logic vector
    %Confusion matrix generated by counting frequencies
    true_negatives = sum((y_pred == 0) .*  (y == 0));
    false_positives = sum((y_pred == 1) .* (y == 0));
    n_0 = true_negatives + false_positives; % = sum(y_test == 0) = # y_test with y = 0
    false_negatives = sum((y_pred == 0) .* (y == 1));
    true_positives = sum((y_pred == 1) .*  (y == 1));
    n_1 = false_negatives + true_positives; % = sum(y_test == 1) = # y_test with y = 1
    confusion_mat = [true_negatives/n_0, false_positives/n_0;
                     false_negatives/n_1, true_positives/n_1];
end
```

Using our previous MLE of β outputs the following confusion matrix: $\begin{bmatrix} 0.7044 & 0.2956 \\ 0.2932 & 0.7068 \end{bmatrix}$

Alternatively, we can use the Matlab build-in command `confusion()`.

Exercise: g) Script below plots the ROC curve of our model (given any extended data set `X_ext` and labels `y`) and also computes the area ($\in$[0.5, 1]) under it:

```
function area = roc_curve(X_ext, y, beta)
    x1 = []; x2 = [];
    %Run through possible values of threshold tau
    for tau = 0:0.01:1
        %Hard predictions with threshold tau and corresponding conf_mat
            [y_pred, confusion_mat] = predicts(y, X_ext, beta, tau);
            %Tracking confusion matrix as tau evolves
            x1 = [confusion_mat(1,2), x1]; %false positives
            x2 = [confusion_mat(2,2), x2]; %true positives
```

```
    end
    plot(x1,x2)
    hold on
    xlabel('False positive rate'); ylabel('True positive rate');
    %Area under the ROC Curve using trapezium rule
    area = trapz(x1, x2);
end
```

For `roc_curve(X_test_ext, y_test, beta)` it returns `area = 0.7314` and the plot (b) in previous page.

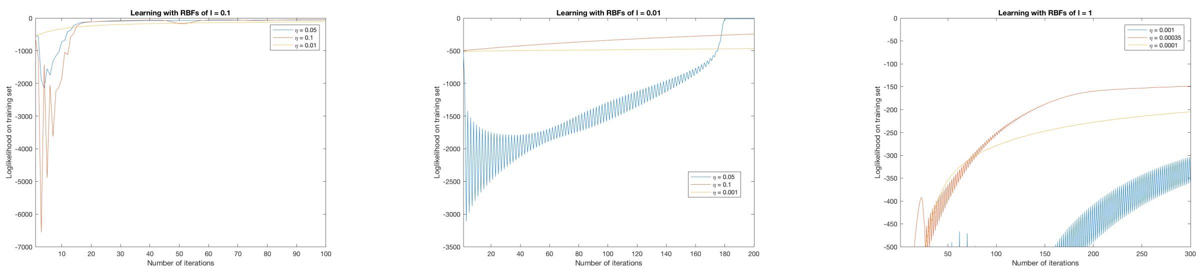
Exercise: h) We define radial basis functions (RBFs) for given width l, in the following way:

$$\tilde{x}^{(1)} = 1 \text{ , } \tilde{x}_n^{(m+1)} = exp(\tfrac{1}{2l^2}\textstyle\sum_{d=1}^{2}(\tilde{x}_n^{(d)} - \tilde{x}_m^{(d)})^2) \text{ for } m = 1, ..., N_{train}$$

Next script computes the extended inputs using RBFs (centred at the training set `X_train` and width `l`) for any data set `X`:

```
function RBFs = RBFs(X, X_train, l)
    N = size(X,1); N_train = size(X_train, 1);
    %Initialise extend radial basis
    RBFs = zeros(N, N_train + 1);          %Each input (1,.., N) has N_train + 1 features
    %Fill the first column with ones
    RBFs(:,1) = ones(N, 1);
    %Use loop to fill remaining entries
    for m = 1:N_train
            for n = 1:N
                RBFs(n,m+1) = exp(-sum((X(n,:)- X_train(m,:)).^2)/(2*l^2));
            end
    end
end
```

Now let's train our Logistic Classification model via `train_logreg(RBF,y_train, learning_rate)` using `RBF = RBFs(X_train, X_train, l)` and `y_train` as inputs. The graphs below show the convergence of the loglikelihood fiex a fixed value of $l \in \{0.01, 0.1, 1\}$ and different η-s



(a) $l = 0.1$ (b) $l = 0.01$ (c) $l = 1$

Exercise: i) To avoid numerical errors, introduce the following patch in `log_likelihood_grad`:

```
var =  X_ext*beta %outside the loop
%...
    if sum_term < -5000
        sum_term = (2*y(n)-1)*(var(n));
    end
```

Where we used the fact that as $\beta^T\tilde{x}_n \to -\infty$,

$$y_n log(\sigma(\beta^T\tilde{x}_n)) + (1 - y_n)log(1 - \sigma(\beta^T\tilde{x}_n)) \sim y_n(\beta^T\tilde{x}_n) + (1 - y_n)(-\beta^T\tilde{x}_n) \sim (2y_n - 1)\beta^T\tilde{x}_n$$

The resulting final training and test loglikelihoods per data point for the different models:
For `l = 0.01`, `loglike_train_pp = -0.1979` `loglike_test_pp = -265.7482`
`area = 0.5744` with confusion matrix: $\begin{bmatrix} 0.9759 & 0.0246 \\ 0.8848 & 0.1152 \end{bmatrix}$
For `l = 0.1`, `loglike_train_pp = -0.0482` `loglike_test_pp = -169.4299`
`area = 0.9163` with confusion matrix: $\begin{bmatrix} 0.9015 & 0.0985 \\ 0.1466 & 0.8534 \end{bmatrix}$
For `l = 1`, `loglike_train_pp = -0.2235` `loglike_test_pp = -106.5136`
`area = 0.9529` with confusion matrix: $\begin{bmatrix} 0.8670 & 0.1330 \\ 0.0838 & 0.9162 \end{bmatrix}$

Both $l = 1, 0.1$ preform well at predicting the labels for the test set (with `tau = 0.5`), whereas $l = 0.01$ does not. Remember l is a measure of the length-scale (or spare) of the extended inputs. Hence for a small l where the inputs are really close to each other we should expected to see soft decision boundaries, giving many wrong predictions. On the other hand, for larger $l = 1$ we should expect hard decision boundaries

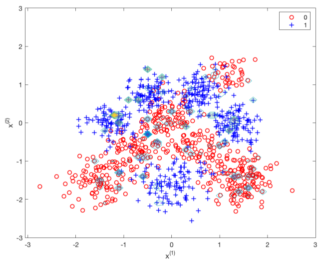
Exercise: j) Introduce a Gaussian prior for the parameters β: $p(\beta^{(m)}) \sim \mathcal{N}(\beta^{(m)}; 0, 1)$ for each of the models in part **(i)**. Then, define the maximum a posteriori (MAP) estimator of β as:

$$\beta_{MAP} := \arg\max_{\beta} p(\beta \mid X, \mathbf{y}) = \arg\max_{\beta} p(\beta \mid X) p(\mathbf{y} \mid X, \beta) \tag{1}$$
$$= \arg\max_{\beta} p(\beta) p(\mathbf{y} \mid X, \beta) = \arg\max_{\beta} log(p(\beta)) + \mathcal{L}(\beta) \tag{2}$$
$$= \arg\max_{\beta} \frac{-\beta^T \beta}{2} + \mathcal{L}(\beta) \tag{3}$$

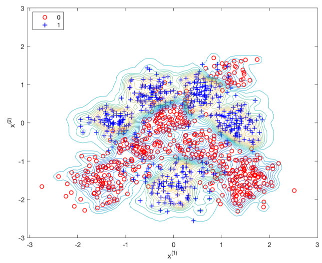
Run gradient ascient for the objective function in line (3) to train our model using the following gradient instead of `log_likelighood_grad`:

```
function grad = posterior_grad(X_ext, y, beta)
N = size(X_ext,1);
sigma = probs_logreg(X_ext, beta);
grad=0;
for n=1:N
    grad_term = (y(n) - sigma(n))*X_ext(n,:)';
    grad = grad + grad_term;
end
grad = grad - beta;
end
```

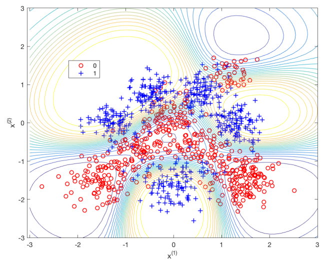
Using the output `beta_map` in `roc_curve` we obtain:
For `l = 0.01`, `loglike_train_pp = -0.1979` `loglike_test_pp = -265.7482`
`area = 0.5631` with confusion matrix: $\begin{bmatrix} 0.9901 & 0.0099 \\ 0.8639 & 0.1361 \end{bmatrix}$
For `l = 0.1`, `area = 0.9454` with confusion matrix: $\begin{bmatrix} 0.9557 & 0.0443 \\ 0.2044 & 0.7966 \end{bmatrix}$
For `l = 1`, `area = 0.9630` with confusion matrix: $\begin{bmatrix} 0.9113 & 0.0887 \\ 0.0890 & 0.9110 \end{bmatrix}$



(a) $l = 0.01$



(b) $l = 0.1$



(c) $l = 1$

So we see that the MAP for β can also do a great job classifying the test data.