1. A majority bin (MB), generative (GG), k-nearest neighbours (KNN) and multi-layer perceptron (MLP) classifiers were implemented (Q2), evaluated and compared in Table 2 (Q3), for classification of 5 different human activities. GG fits Gaussian prior probability distributions of the form $p(\mathbf{x}|C_k) = a \exp{(\mathbf{x}-\mu)^T \Sigma_k^{-1} (\mathbf{x}-\mu)}$ to each class C_k using a training dataset X^p and classifies test datapoint $\mathbf{x^p}$ as the class C_k with largest posterior probability $p(C_k|\mathbf{x}) = \frac{p(\mathbf{x}|C_k)p(C_k)}{p(\mathbf{x})}$. Our KNN method assigns class membership under majority voting among the K-nearest neighbours to $\mathbf{x^p}$, where votes are weighted by the inverse of the distance of the neighbour to $\mathbf{x^p}$. This weighted voting method avoids classification indecision that would result from a possible tie between two or more classes among the K nearest neighbours of $\mathbf{x^p}$ within an non-weighted voting system. MLP is a neural network with 2 hidden layers comprising 40 units each with hyperbolic tangent activation functions. It uses the generalized delta rule with backward propagation to adjust network weights, as well as L2 loss $E = \frac{1}{2} \sum_i (\mathbf{y_i^p} - \mathbf{V_i^M})^2$ as an error measure, where y^p is the desired network output for training example $\mathbf{x^p}$ and $\mathbf{V^M}$ is the predicted output. For MB, since our feature space is 64-dimensional, N bins per dimension would require a minimum of N^{64} datapoints for the algorithm to be implemented. However, we only have $64,000$ datapoints where $N^{64} \gg 64,000$ for $N \geq 2, N \in \mathbb{N}$, hence **MB is rejected**.

The dataset was visualized following a brute-force approach where class-specific empirical distributions were plotted for each feature to identify class-specific trends (**1**) and assess class separability (**2**) to guide our choice of classification pipeline (**3**). We observe (**1**) that class 1 is generally multi-modal while other classes follow a skewed Gaussian or Poisson distribution (Table 1), hence we expect poor generalization of GG for class 1 as we are fitting a unimodal model to multimodal data. This may be mitigated by the generally good separability (**2**) of classes 1 and 2: their distribution, across features 1 to 60 (esp. 2, 5-6, 10, 13-14, 56, 58-60), rarely overlap with the distribution of any other class, while conversely classes 4 and 5 often overlap. Besides, we note that features 61 to 64 are poorly informative since datapoints are normally distributed around the same mean irrespective of the class. We therefore choose to **discard these features** in our classification pipeline as they result in additional system noise. Similarly, we can here note that if model simplicity and computational time were primary design objectives, we may consider ignoring the following redundant classes, for which the magnitude of the correlation coefficient with another feature exceeds 0.9: $4, 5, 6, 7, 10, 13, 18, 27, 30, 42$.

Two pre-processing stages are applied to the training data (**3**): feature normalization and class balancing, defined here with special relevance to each classifier. **Feature normalization** is performed by **z-scoring** the training data, such that for each feature the mean value across training examples is 0 and standard deviation is 1, using $x_{out} = \frac{x_{in} - \mu_{in}}{\sigma_{in}}$, where μ_{in} and σ_{in} are stored and applied to the test data through the same transformation, such as to not introduce any bias. Centering all features on 0 prevents potentially unstable large bias terms in MLP, while scaling by the standard deviation prevents numerical explosion of the weights. It also ensures that all features contribute equally to the classification output in the case of KNN, given that the distance from a test data point to another training example is *a priori* of the same scale across all features/dimensions, and ensures numerical stability in the case of GG, since the exponential term in the expression of a multivariate Gaussian $p(x|C_k) = a \exp{(\mathbf{x}-\mu)^T \Sigma_k^{-1} (\mathbf{x}-\mu)}$ aggressively pushes $p(x|C_k)$ to ∞ or 0 if the term in the exponential is of large magnitude (which z-scoring avoids). More generally, scaling renders the features unit-independent.

Class balancing ensures that the training dataset contains equal number of datapoints for each class, which limits classification bias. Notably, Σ_k varies with sample size in GG, and similarly as K approaches the number of training examples belonging to class k, the bias in assigning membership to k with KNN becomes increasingly biased against this class. Class balancing was performed by extracting the number of training datapoints N_t for the least represented class and under-sampling the training dataset such that each class would be be represented by N_t datapoints. This technique is fast and easy to implement, however in the unlikely case where

one class would be represented by exactly 1 datapoint, the Gaussian generative approach would fail, since extraction of a covariance matrix requires at least 2 datapoints (if $K < 5$, where 5 is the number of classes, then KNN would also fail). In the case of the KNN, we have chosen to implement class-balancing via a weighting of the vote of each datapoint by a class-dependant factor α_k reflecting class imbalance, such that α_k equals the ratio of the number of training datapoints for the most represented class to the number of training datapoints for class C_k.

Class	**1**	**2**	**3**	**4**	**5**
Shape (% features)	Multimodal (89%)	Gaussian (76%)	Gaussian (90%)	Gaussian (94%)	Gaussian (94%)
Separability (relative)	Excellent	Excellent	Good	Poor	Poor

Table 1: **Preliminary analysis of class-specific empirical trends in the unprocessed dataset**

2. GG, KNN and MLP classifiers (described in the first paragraph of Question 1) were implemented, along with a GUI allowing for the selection of pre-processing steps (see appendix). Training algorithms were implemented for the parametric approaches: GG (class-specific mean μ_k and covariance Σ_k are trained) and MLP (weight matrix w and biases B are trained).

3. The classifiers were tested and validated using **10-fold cross-validation**, using **mean and standard deviation in test accuracy** as performance parameters, along with the **confusion matrix**. During every 1 of the 10 validation tests 90% of the available data is used as training, such as to ensure good generalization properties, and the classifier performance is evaluated on the remaining 10% of the dataset, such as to limit bias in the estimation of classification error probability. We hence slightly value generalization over accuracy estimation bias.

The **choice of hyperparameters** was determined experimentally, taking maximization of mean accuracy and reduction of accuracy variability as performance parameters, leading to the choice $K = 5$ for KNN. Although higher mean accuracy was observed for $K < 5$, reducing K sacrifices resistance to outliers noise, hence we have chosen to settle for the plateau region at $K = 5$ to ensure good generalization on noisy datasets (Figure 1). In addition, this ensures that the algorithm would not fail even in the unlikely scenario where the training dataset would contain only one example of each class. Similarly, we select $\eta = 0.002$ and $\beta = 0.6$ for the learning rate and activation gain of the units in MLP as they maximize test accuracy (Figure 2 and 3). The classifier performance was not improved when using a time-dependant learning rate in the form $\eta(t) = \frac{\eta_0}{t}$, where t is the number of epochs (classification accuracy under 10-fold cross-validation for $\eta_0 = 0.002$ of 94.1417 ± 0.9395, compare against Table 2).

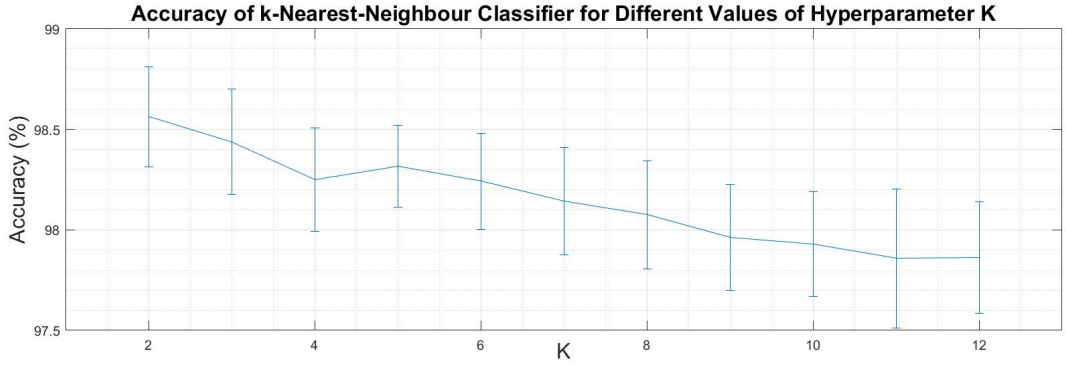

Figure 1: **KNN accuracy under 10-fold cross-validation against K, highlighting optimal trade-off between smoothing effect of high K, minimal variance and maximal mean accuracy at $K = 5$**

The classifiers were evaluated to **settle our choice of preferred classifier**. Overall, KNN exhibits highest mean test accuracy (owing to large size of the training dataset) and lowest accuracy variance. In addition, KNN exhibits robustness to noise in training data resulting from the "smoothing" effect of increasing K, and benefits from the high quality of the available feature set which offers good class separability. Its computational cost is however high (n distance computation in 64-dimensions per test datapoint $\mathbf{x}$, where n is the number of training datapoints).

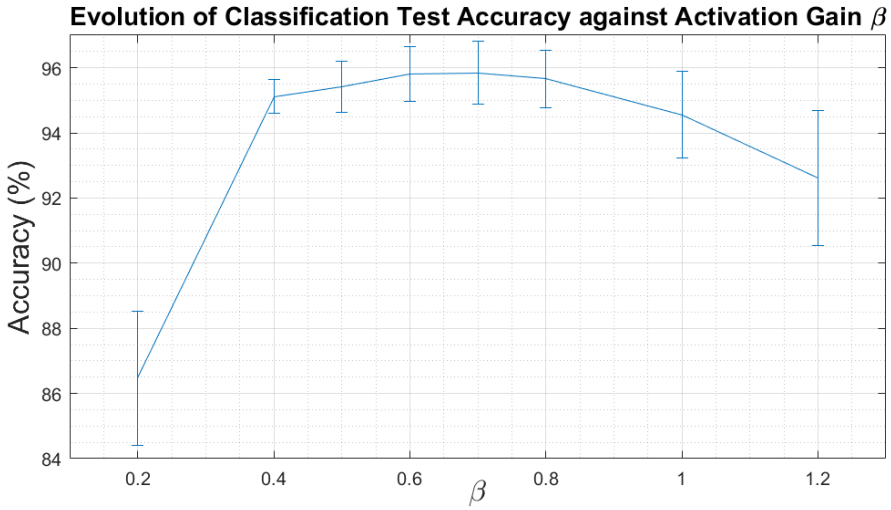

Figure 2: **MLP accuracy under 10-fold cross-validation against β for $\eta = 0.002$, highlighting optimal trade-off between minimal accuracy variance and maximal mean accuracy at $\beta = 0.6$**

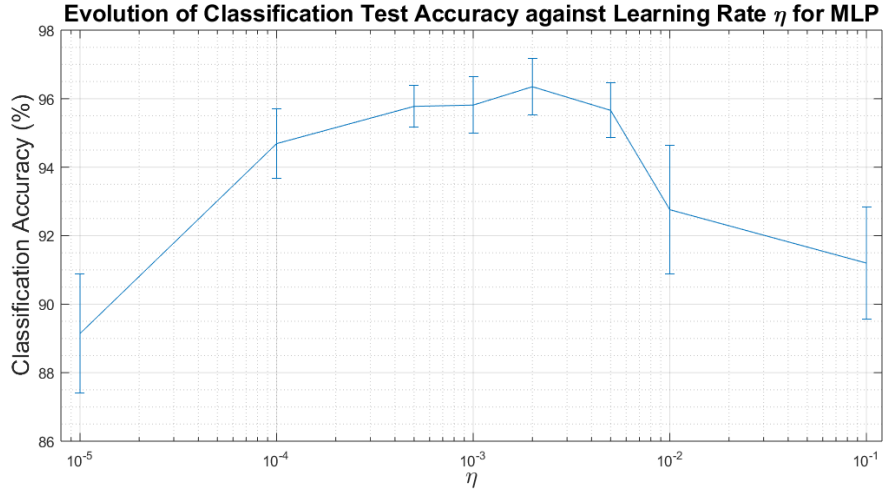

Figure 3: **MLP accuracy under 10-fold cross-validation against η for $\beta = 0.6$, highlighting optimal trade-off between minimal accuracy variance and maximal mean accuracy at $\eta = 0.002$**

Conversely, GG benefits in the classification step from independence of computational cost on n, however it generalizes more poorly than KNN due to the generally multimodal distribution of class 1 datapoints across features. Lastly, MLP benefits from increased model flexibility with two hidden layers enabling for fine tuning of nonlinear decision boundaries, which comes at the cost of highest computational load. MLP is expected to perform better than KNN/GG in the case of poorly separable classes, however owing to the advantageous feature set available for this classification problem, we observe that it is outperformed by KNN and GG. Besides,the confusion matrices indicate that MLP performs poorly on classification of class 5, with an error rate of 15.6379%. Overall, the error rate across classifiers is higher for classes 4 and 5, which is a consequence of the poor separability observed for these classes and discussed in Question 1. Consequently, **KNN is selected as our preferred classifier**, since it optimizes our design parameters of computational time minimization and generalization maximization expressed through the mean classification test accuracy.

Algorithm	**GG**					**KNN**					**MLP**				
Accuracy (%)	96.9625 ± 0.2549					98.2167 ± 0.2892					96.6042 ± 0.5275				
Confusion matrix (%)	99.9824	0.0176	0	0	0	99.2167	0.8095	0	0	0	99.7921	0.1906	0	0.0173	0
	0	.7658	0.2342	0	0	0.1946	99.4487	0.2757	0	0.0811	0.0486	99.4820	0.3885	0.0162	0.0647
	0.0549	3.0569	96.6502	0.1098	0.1281	0.0725	0.7254	97.5517	0.8524	0.7980	0.2359	1.1432	97.3508	0.7984	0.4718
	2.3035	0.2612	1.7098	91.948	4.6307	0	0	0	96.7383	3.2617	0	0	0.1656	95.3366	4.4978
	0.0748	2.8432	1.1223	4.2649	91.6947	0	0	0	3.2447	96.7553	0	0	0.0343	15.6036	84.3621

Table 2: **Comparison of generative Gaussian (GG), k-nearest neighbours (KNN) and multilayer perceptron (MLP) classifiers showing classification accuracy and confusion matrices, evaluated through 10-fold cross-validation**

1 Appendix

The fully annotated source code which generated the results discussed above is included below. This includes, in order: the function `LookatData.m` used to visualize the class-specific distribution of the dataset for each features (press any key to progress to the next feature), the GUI `Main.m`, used to select pre-processing steps as well as classifiers to be implemented, and computing classification accuracy for each trial through n-fold cross-validation as well as the confusion matrix accumulated over the n trials, and finally the training and classifying functions for GG, MLP and KNN, respectively named `TrainsClassifierGCA.m`, `TrainsClassifierMLP`, `TrainClassifierX.m` `ClassifyGCA.m`, `ClassifyMLP` and `ClassifyX.m`. Please note that only `Main.m` has to be run to access all training functions and classifiers.

```
%% COURSEWORK 2: HUMAN ACTIVITY RECOGNITION
%%% This is a function for preliminary visualization of the dataset.
%%% Bastien CABA, MEng Y4, CID: 01060785

%% IMPORT DATA
clearvars; clc; close all; load('data.mat');     %Clear workspace and command window, load data
labels_raw = data(:,1);                          %Matrix containing features (columns) of datapoints (rows)
features_raw = data(:,2:length(data(1,:)));      %Column vector containing label of datapoints (rows)

%% SEPARATE FEATURE VALUES OF DATASET INTO INTO CLASSES AND PLOT CLASS-DEPENDANT HISTOGRAM FOR EACH FEATURE
for feat = 1:64   %Iterate over features
    for i = 1:5   %Iterate over classes
        %Select datapoints belonging to this class
        X{i} = features_raw((labels_raw == i),feat);
    end
    fprintf('Press any key to progress to the next feature.');
    pause;
    figure;
    %Display class-specific distribution for each feature
    histogram(X{1}); hold on;
    histogram(X{2}); histogram(X{3});
    histogram(X{4}); histogram(X{5});
    legend('Class 1', 'Class 2', 'Class 3', 'Class 4', 'Class 5');
    xlabel('Feature value'); ylabel('Number of occurences');
    title(['Class-specific data distributions for feature ', num2str(feat)]);
    clc;
end

pause;
fprintf('All features have been visualized.');
clc; close all;
```

```
%% COURSEWORK 2: HUMAN ACTIVITY RECOGNITION
%%% This is a GUI guiding through the classification pipeline.
%%% Bastien CABA, MEng Y4, CID: 01060785

%% IMPORT AND CROP DATA
clearvars; clc; close all; load('data.mat');     %Clear workspace and command window, load data
labels_raw = data(:,1);                          %Matrix containing features (columns) of datapoints (rows)
features_raw = data(:,2:length(data(1,:)));      %Column vector containing label of datapoints (rows)

%% SPLIT DATA IN TRAINING AND TESTING SETS FOR N-FOLD VALIDATION
% Get user input for k-cross validation
fprintf('The complete dataset will first be divided into a training subset and a testing subset.\n');
prompt = 'Select K for K-fold cross-validation > ';
n = input(prompt);
clear prompt;

% Split the data
[data_subsets] = split_data(features_raw, labels_raw, n);
fprintf('The input data has been randomly partitioned into %d subsets in variable data_subsets.\n', n);

%% SELECT WHICH CODE TO RUN
% KNN
prompt = '\nDo you wish to run KNN (0:NO, 1:YES)?  > ';
knn_ctrl = input(prompt);

% GG
prompt = 'Do you wish to run GG (0:NO, 1:YES)?  > ';
gg_ctrl = input(prompt);

% MLP
prompt = 'Do you wish to run MLP (0:NO, 1:YES)?  > ';
mlp_ctrl = input(prompt);

%% SELECT HYPERPARAMETER K for KNN (not applied since here to fit the classifier format expected)
% if(knn_ctrl == 1)
%     % Get user input
%     prompt = '\nSelect K for K-nearest-neighbour > ';
%     k_knn = input(prompt);
%
%     % Pre-processing pipeline
%     prompt = 'Do you wish to z-score the training data for KNN (0:NO, 1:YES)? > ';
%     standard_knn = input(prompt);
%     prompt = 'Do you wish to balance the training data for KNN (0:NO, 1:YES)? > ';
%     balance_knn = input(prompt);
% end
```

```
%% PRE-PROCESSING for GG
if(gg_ctrl == 1)
    prompt = '\nDo you wish to z-score the training data for GG (0:NO, 1:YES)? > ';
    standard_gg = input(prompt);

    prompt = 'Do you wish to balance the training data for GG (0:NO, 1:YES)? > ';
    balance_gg = input(prompt);

    fprintf('The parameters for generative approach will now be trained.\n');
    fprintf('A Gaussian model is assumed for each class.\n');
end

%% PRE-PROCESSING for MLP
if(mlp_ctrl == 1)
    prompt = '\nDo you wish to z-score the training data for MLP (0:NO, 1:YES)? >↙
';
    standard_mlp = input(prompt);

    prompt = 'Do you wish to balance the training data for MLP (0:NO, 1:YES)? > ';
    balance_mlp = input(prompt);
end

%% N-FOLD CROSS VALIDATION
for i = 1:n
    %% RESET TRAINING SETS
    train_input = []; train_output = [];

    %% SELECT TRAINING AND TEST DATASETS
    for j = 1:n
        if(j == i)
            test_input = data_subsets{j,1};      %One of the n subsets becomes the↙
new test set
            test_output = data_subsets{j,2};     %Associate corresponding true↙
labels
        else
            train_input = [train_input; data_subsets{j,1}];      %Other subsets are↙
training sets
            train_output = [train_output; data_subsets{j,2}];    %Associate↙
corresponding labels
        end
    end

    %% TRAIN THE CLASSIFIERS AND CLASSIFY THE TEST INPUT
    %% Generative Gaussian
    if(gg_ctrl == 1)
        % Train the parameters (class mean and class covariance matrix)
        parametersGCA = TrainsClassifierGCA(train_input, train_output, standard_gg,↙
balance_gg);

        % Classify the test dataset
        classGG = ClassifyGCA(test_input, parametersGCA);
```

```
        % Compute classification accuracy and confusion matrix
        [accuracyGG(i), confGG{i}] = accuracy(classGG, test_output);
    end

    %% Multilayer Perceptron
    if(mlp_ctrl == 1)
        % Train the parameters (weigths and bias in network)
        parametersMLP = TrainsClassifierMLP(train_input, train_output, ↙
standard_mlp, balance_mlp);

        % Classify the test dataset
        classMLP = ClassifyMLP(test_input, parametersMLP);

        % Compute classification accuracy and confusion matrix
        [accuracyMLP(i), confMLP{i}] = accuracy(classMLP, test_output);
    end

    %% K-nearest neighbours
    if(knn_ctrl == 1)
        % Train the parameters for KNN
        parametersKNN = TrainClassifierX(train_input, train_output);

        % Classify the test dataset
        classKNN = ClassifyX(test_input, parametersKNN);

        % Compute classification accuracy and confusion matrix
        [accuracyKNN(i), confKNN{i}] = accuracy(classKNN, test_output);
    end
end

%% NORMALIZE THE CONFUSION MATRICES
%% Generative Gaussian
if(gg_ctrl == 1)
    %Display test classification accuracy
    fprintf('\nThe mean classification accuracy on test data for the generative ↙
Gaussian approach evaluated through %d-fold cross-validation is of %f, with %f ↙
standard deviation.', n, mean(accuracyGG), std(accuracyGG));

    % Initialize an empty confusion matrix of correct dimensions
    confusionGG = zeros(length(confGG{1}(:,1)), length(confGG{1}(1,:)));

    % Sum confusion matrices over all cross-validation trials
    for i = 1:n
        confusionGG = confusionGG + confGG{i};
    end

    % Normalize each term over total number of actual occurences
    for j = 1:length(confGG{1}(:,1))
        norm = sum(confusionGG(j,:));
        confusionGG(j,:) = (confusionGG(j,:)/norm)*100;
    end

    % Display confusion matrix
```

```
    fprintf('\nThe confusion matrix accumulated over all trials of cross-validation for GG is given below:\n');
    confusionGG
end

%% K-nearest neighbours
if(knn_ctrl == 1)
    %Display test classification accuracy
    fprintf('\nThe mean classification accuracy on test data for the k-nearest neighbours approach evaluated through %d-fold cross-validation is of %f, with %f standard deviation.', n, mean(accuracyKNN), std(accuracyKNN));

    % Initialize an empty confusion matrix of correct dimensions
    confusionKNN = zeros(length(confKNN{1}(:,1)), length(confKNN{1}(1,:)));

    % Sum confusion matrices over all cross-validation trials
    for i = 1:n
        confusionKNN = confusionKNN + confKNN{i};
    end

    % Normalize each term over total number of actual occurences
    for j = 1:length(confKNN{1}(:,1))
        norm = sum(confusionKNN(j,:));
        confusionKNN(j,:) = (confusionKNN(j,:)/norm)*100;
    end

    % Display confusion matrix
    fprintf('\nThe confusion matrix accumulated over all trials of cross-validation for KNN is given below:\n');
    confusionKNN
end

%% Multilayer perceptron
if(mlp_ctrl == 1)
    %Display test classification accuracy
    fprintf('\nThe mean classification accuracy on test data for the multi-layer perceptron approach evaluated through %d-fold cross-validation is of %f, with %f standard deviation.', n, mean(accuracyMLP), std(accuracyMLP));

    % Initialize an empty confusion matrix of correct dimensions
    confusionMLP = zeros(length(confMLP{1}(:,1)), length(confMLP{1}(1,:)));

    % Sum confusion matrices over all cross-validation trials
    for i = 1:n
        confusionMLP = confusionMLP + confMLP{i};
    end

    % Normalize each term over total number of actual occurences
    for j = 1:length(confMLP{1}(:,1))
        norm = sum(confusionMLP(j,:));
        confusionMLP(j,:) = (confusionMLP(j,:)/norm)*100;
    end
```

```
    % Display confusion matrix
    fprintf('\nThe confusion matrix accumulated over all trials of cross-validation for MLP is given below:\n');
    confusionMLP
end

%% LIST OF SUB-FUNCTIONS
%% SPLITTING DATA IN N EQUALLY-SIZED SUBSETS
function [list_subsets] = split_data(data_input, data_output, n)
%% INITIALIZATION OF SUBSET LENGTH
len_in = length(data_input);        %Number of datapoints in complete set
set_size = floor(len_in/n);         %Number of datapoints needed in each subset
elems = randperm(len_in)';          %Randomly shuffle data indices

%% CREATE N INDICES SUBSETS
for i = 1:n
    temp_lim = (i-1)*set_size+1:i*set_size;     %Boundaries element indices for each subset
    data_list{i} = elems(temp_lim);             %List of elements in each of the n subsets
end

%% APPEND REMAINING INDICES TO LAST SUBSET
data_list{n} = [data_list{n}; elems(n*set_size+1:length(data_input))];

%% CREATE N DATA SUBSETS
for i = 1:n
    list_subsets{i,1} = data_input(data_list{i},:);
    list_subsets{i,2} = data_output(data_list{i},:);
end
end

%% CLASSIFICATION TEST ACCURACY COMPUTATION
function [performance, conf] = accuracy(actual, predicted)
% Initialize variables
count = 0; conf = [];

% Count number of correct predictions
for i = 1:length(actual)
    if(actual(i) == predicted(i))
        count = count + 1;
    end
end

% Compute percentage of correct predictions and confusion matrix
performance = (count/length(predicted))*100;
conf = confmat(actual, predicted);
end


%% CONFUSION MATRIX COMPUTATION
function [confusion] = confmat(vec_actual, vec_predicted)
% Indentify classes present in predicted and actual vectors
```

```
classes_actual = unique(vec_actual);
classes_predicted = unique(vec_predicted);

% Initialize confusion matrix
confusion = zeros(length(classes_actual), length(classes_predicted));

% Compute number of actual/predicted class occurences
for i = 1:length(vec_predicted)
    confusion(vec_actual(i), vec_predicted(i)) = confusion(vec_actual(i), ↙
vec_predicted(i)) + 1;
end
end
```

```
%% TRAINING PARAMETERS FOR GCA (generative classification approach)
    %%% We define one class probability distribution model per class (Gaussian model assumed)
        %%% Each class probability distribution model has 2 parameters (mean and covariance)

% IN: "inputs" is a matrix containing features (columns) of datapoints (rows)
% IN: "outputs" is a column vector containing the class of each data point (rows)
% IN: "standard" is a boolean control determining whether data should be z-scored
% IN: "balance" is a boolean control determining whether the testing set should be class-balanced

function parameters = TrainsClassifierGCA(inputs, outputs, standard, balance)
%% STEP 0: REMOVE NON-INFORMATIVE FEATURES
inputs = inputs(:,1:60);    %Remove features 61 to 64

%% EXTRACT DATA INFO
info = data_info(inputs, outputs);
class_names = info{4};      %Vector containing class names
num_classes = info{3};      %Number of classes in test set
num_features = info{2};     %Number of features in datapoints from test set
num_datapoints = info{1};   %Number of datapoints in test set

%% Z-SCORING (PRE-PROCESSING STEP)
if(standard == 1)
    [inputs, param_z] = z_standard(inputs);
    fprintf('The data was z-scored.\n');
else
    fprintf('The data was not z-scored.\n');
    [temp, param_z] = z_standard(inputs);
    clear temp;
end

%% BALANCING (PRE-PROCESSING STEP)
if(balance == 1)
    [inputs, outputs, num_datapoints] = balances(inputs, outputs, info);
    fprintf('The data was balanced.\n');
else
    fprintf('The data was not balanced.\n');
end
clear balance;

%% INITIALIZE PARAMETERS
parameters{1} = zeros(num_classes, num_features);                   %Feature means (column) per class (row)
parameters{2} = zeros(num_features, num_features, num_classes);     %Inverse covariance per class
parameters{3} = class_names;                                        %Class names
parameters{4} = zeros(1, num_classes);                              %Class prior probabilities
parameters{5} = repmat(inv(cov(inputs)),[1 1 num_classes]);         %Covariance matrix computed over all input data
parameters{6} = standard;                                           %Indicates if
```

```
inputs were z-scored
parameters{7} = param_z;                                          %Parameters for
z-score

p_class = zeros(1, num_classes);                                  %Temporary
class probability
means_temp = zeros(num_classes, num_features);                    %Temporary
means
invcov_temp = zeros(num_features, num_features, num_classes);     %Temporary
inverse covariance

%% COMPUTE MEAN AND COVARIANCE
for i = 1:num_classes   %Iterate over classes
    %Select datapoints belonging to this class
    X = inputs((outputs == class_names(i)),:);
    fprintf('The training dataset contains %d points that belong to class %d.\n',
length(X), class_names(i));

    %Compute class mean and inverse covariance for feature across datapts
    means_temp(i,:) = mean(X);          %Mean
    invcov_temp(:,:,i) = inv(cov(X));   %Inverse covariance

    %Probability of observing each class
    p_class(i) = length(X)/num_datapoints;
end

%% SET PARAMETERS
parameters{1} = means_temp;     %Class-specific mean across features
parameters{2} = invcov_temp;    %Class-specific inverse covariance matrix
parameters{4} = p_class;        %Prior probability of each class

end


function [info] = data_info(inputs, outputs)
%% EXTRACT BASIC DATA INFO
[num_datapts, num_features] = size(inputs);         %Number of datapoints and
number of features
class_names = unique(outputs)';                     %Name of classes
num_class = length(class_names);                    %Number of classes

%% DISPLAY DATA INFO
fprintf('The complete dataset contains %d data points with %d features.\n',
num_datapts, num_features);
class_string = sprintf('%d ', class_names);
fprintf('There are %d different classes observed in the training dataset, which can
be: %s\n\n', num_class, class_string);

%Return an array with information on data
info = {num_datapts, num_features, num_class, class_names};
end

%% FUNCTION TO Z-SCORE DATA
```

```
function [feat_z, param_z] = z_standard(inputs)
%% Z-SCORE DATA, STORE PARAMETERS
[feat_z, mu, sigma] = zscore(inputs);
param_z = [mu; sigma];
end

%% FUNCTION TO CLASS-BALANCE A SET
function [feat_bal, label_bal, num_datapts_bal] = balances(inputs, outputs, info)
%% BALANCE INPUTS SO EACH CLASS IS EQUALLY REPRESENTED
class_names = info{4};
num_class = info{3};

count_class = zeros(1,num_class);   %Count number of elements per class in dataset
original_length = length(outputs);  %Original length of test dataset
for i = 1:num_class
    temp = inputs((outputs == class_names(i)),:);
    count_class(i) = length(temp);
end

%% SELECT NUM OF ELEM IN LEAST REPRESENTED CLASS AND BALANCE INPUTS
min_length = min(count_class);      %Number of elements in least represented class
bal_inputs = []; bal_outputs = [];  %Initialize new empty sets

for i = 1:num_class     %Iterate over classes
    order_inputs = inputs((outputs == class_names(i)),:);           %Select inputs with that class
    bal_inputs = [bal_inputs; order_inputs(1:min_length,:)];        %Select only min_length datapoints of that class
    bal_outputs = [bal_outputs; class_names(i)*ones(min_length,1)]; %Associate corresponding labels
end

feat_bal = bal_inputs;                  %OUTPUT: Original dataset cropped so that every class has as many datapoints as the least represented class
label_bal = bal_outputs;                %OUTPUT: Corresponding labels
num_datapts_bal = length(label_bal);    %New size of the training set

%% DISPLAY NEW INFORMATION
fprintf('The training dataset now contains %d datapoints, reduced from the original %d.\n', num_datapts_bal, original_length);
end
```

```
%% Training Perceptron Learning
%%% This function trains the parameters for multilayer perceptron classification
%%% The parameters are network weights which best separate the training data into classes

%%% The input variable "inputs" is a matrix containing features (second dimension) of data points (first dimension)
%%% The input variable "outputs" is a column vector containing the class of each data point

function parameters = TrainsClassifierMLP(inputs, outputs, standard, balance)
%% STEP 0: REMOVE NON-INFORMATIVE FEATURES
inputs = inputs(:,1:60);    %Remove features 61 to 64

%% EXTRACT DATA INFO
info = data_info(inputs, outputs);
class_names = info{4};      %Vector containing name of classes
num_classes = info{3};      %Number of classes represented in training set
num_features = info{2};     %Number of features for each datapoint
num_datapoints = info{1};   %Number of datapoints in training set

%% SELECT IF DATA MUST BE BALANCED AND/OR Z-SCORED
fprintf('The parameters for multilayer perceptron learning will now be trained.\n');
if(standard == 1)
    [inputs, param_z] = z_standard(inputs);
    fprintf('The data was z-scored.\n');
else
    fprintf('The data was not z-scored.\n');
    [temp, param_z] = z_standard(inputs);
    clear temp;
end

if(balance == 1)
    [inputs, outputs, num_datapoints] = balances(inputs, outputs, info);
    fprintf('The data was balanced.\n');
else
    fprintf('The data was not balanced.\n');
end
clear balance;

%% INITIALIZE HYPERPARAMETERS
nabla = 0.002;  %Learning rate (not called eta to limit chances of mistaking it with beta)
beta = 0.6;     %Activation gain
nbrOfNodes = [num_features, 40, 40, num_classes];    %Nodes per layer
nbrOfLayers = length(nbrOfNodes);                    %Number of layers
tol = 0.0001; error = tol + 1;  %Tolerance in the network output error
epoch = 0; max_epoch = 200;     %Max epochs if weights have not converged

%%% Initialize random weight matrices with small magnitude terms
%%% Weights are all between -1 and +1
%%% E.G. Weight matrix 1 connects layer 1 to 2
```

```
for m = 1:(nbrOfLayers-1)
    w{m} = rand(nbrOfNodes(m+1), nbrOfNodes(m)) - rand(nbrOfNodes(m+1), nbrOfNodes(m));
    B{m} = rand(nbrOfNodes(m+1),1) - rand(nbrOfNodes(m+1),1);
end

%% FORWARD AND BACKWARD PASS FOR EACH TRAINING POINT
while((error>tol)&&(epoch<max_epoch))    %Check for weights convergence and number of epochs
    epoch = epoch + 1; error = 0;        %Reset error, increment epoch
    for training = 1:num_datapoints      %One epoch is a sweep on all datapoints
        LayerInput{1} = inputs(training,:)';    %Input to the first layer is a new test datapoint
        ActualOutput = outputs(training);       %Associate corresponding label

        % Compute desired network output
        TargetOut = zeros(num_classes, 1);  %Initialiwe desired output
        for class = 1:num_classes
            if(class == ActualOutput)
                TargetOut(class) = 0.95;    %Network should output close to +1 for the neuron associated with the correct class
            else
                TargetOut(class) = -0.95;   %Network should output clsoe to -1 for the neurons associated with any other class
            end
        end

        %% FORWARD PASS
        for layer = 1:(nbrOfLayers-1)   %Iterate over layers
            input = LayerInput{layer};  %Select input to that layer
            weights = w{layer};         %Select weights connecting that layer to the next
            u{layer+1} = weights*input + B{layer};                  %Compute activity of next layer
            LayerOutput{layer+1} = activation(u{layer+1}, beta);    %Compute activation of next layer
            LayerInput{layer+1} = LayerOutput{layer+1};             %Activation of one layer is input to the next
        end

        %% COMPUTE ERROR
        output = LayerOutput{nbrOfLayers};              %Network output is the output of the last layer
        error = (1/2)*(sum((TargetOut - output).^2));   %Compute L2 loss between predicted and actual output

        %% BACKWARD PASS (Generalized delta rule)
        Delta{nbrOfLayers} = deriv_activation(u{nbrOfLayers}, beta).*(TargetOut-output);    %COmpute deltas for output layer
        for layer = (nbrOfLayers-1):-1:2                            %Iterate over layers backwards
            gradient = deriv_activation(u{layer}, beta);            %Compute activation function gradient at the unit activity for each layer
```

```
            Delta{layer} =  gradient.*(Delta{layer+1}'*w{layer})';   %Back-propagate errors
        end

        %% UPDATE WEIGHTS
        for Layer = 1:nbrOfLayers-1
            w{layer} = w{layer} + nabla*(Delta{layer+1}*LayerInput{layer}');
            B{layer} = B{layer} + nabla*Delta{layer+1};
        end
    end
end

%% Parameters
parameters{1} = w;              %Weight matrix
parameters{2} = class_names;    %Vector of class names
parameters{3} = param_z;        %Parameters for z-scoring
parameters{4} = standard;       %Control indicating whether test data should be z-scored
parameters{5} = nbrOfLayers;    %Number of network layers
parameters{6} = beta;           %Activation gain beta
parameters{7} = B;              %Network biases

end

%% NEURON ACTIVATION FUNCTION
function [output] = activation(input, beta)
output = tanh((beta/2)*input);
end

%% GRADIENT OF NEURON ACTIVATION FUNCTION
function [output] = deriv_activation(input, beta)
output = (beta/2)*(1-(tanh((beta/2)*input).^2));
end

%% PREPROCESSING FUNCTIONS
function [info] = data_info(inputs, outputs)
%% EXTRACT BASIC DATA INFO
[num_datapts, num_features] = size(inputs);         %Number of datapoints and number of features
class_names = unique(outputs)';                     %Name of classes
num_class = length(class_names);                    %Number of classes

%% DISPLAY DATA INFO
fprintf('The complete dataset contains %d data points with %d features.\n', num_datapts, num_features);
class_string = sprintf('%d ', class_names);
fprintf('There are %d different classes observed in the training dataset, which can be: %s\n\n', num_class, class_string);

info = {num_datapts, num_features, num_class, class_names};
end
```

```
function [feat_z, param_z] = z_standard(inputs)
%% Z-SCORE DATA, STORE PARAMETERS
[feat_z, mu, sigma] = zscore(inputs);
param_z = [mu; sigma];
end


%% FUNCTION TO CLASS-BALANCE A SET
function [feat_bal, label_bal, num_datapts_bal] = balances(inputs, outputs, info)
%% BALANCE INPUTS SO EACH CLASS IS EQUALLY REPRESENTED
class_names = info{4};
num_class = info{3};

count_class = zeros(1,num_class);    %Count number of elements per class in dataset
original_length = length(outputs);   %Original length of test dataset
for i = 1:num_class
    temp = inputs((outputs == class_names(i)),:);
    count_class(i) = length(temp);
end

%% SELECT NUM OF ELEM IN LEAST REPRESENTED CLASS AND BALANCE INPUTS
min_length = min(count_class);       %Number of elements in least represented class
bal_inputs = []; bal_outputs = [];   %Initialize new empty sets

for i = 1:num_class      %Iterate over classes
    order_inputs = inputs((outputs == class_names(i)),:);           %Select inputs with that class
    bal_inputs = [bal_inputs; order_inputs(1:min_length,:)];        %Select only min_length datapoints of that class
    bal_outputs = [bal_outputs; class_names(i)*ones(min_length,1)]; %Associate corresponding labels
end

feat_bal = bal_inputs;                  %OUTPUT: Original dataset cropped so that every class has as many datapoints as the least represented class
label_bal = bal_outputs;                %OUTPUT: Corresponding labels
num_datapts_bal = length(label_bal);    %New size of the training set

%% DISPLAY NEW INFORMATION
fprintf('The training dataset now contains %d datapoints, reduced from the original %d.\n', num_datapts_bal, original_length);
end
```

```
%% TRAINING PARAMETERS KNN (k-nearest-neighbour)
%%% 1. PRE-PROCESSING: Remove non-informative features 61-64
%%% 2. PRE-PROCESSING: Z-score the training data
%%% 3. PRE-PROCESSING: Store mean and standard deviation to Z-score the testing data in the same way
%%% 4. PRE-PROCESSING: Compute coefficient alpha accounting for class imbalance

%%% IN: "input" is a matrix containing the training datapoints as rows
%%% IN: "label" is a vector containing the class of each training datapoint
%%% OUT: "parameters" is an array containg trained parameters for KNN classification

function [parameters] = TrainClassifierX(input, label)
%% STEP 0: REMOVE NON-INFORMATIVE FEATURES
input = input(:,1:60);              %Remove features 61 to 64

%% STEP 1: EXTRACT DATA INFO
info = data_info(input, label);     %Extracts/displays training dataset information

%% STEP 2: DATA PREPROCESSING
% Z-SCORING TRAINING SET AND STORING PARAMETERS
[input, param_z] = standardize(input);
clear standard; fprintf('The data was z-scored.\n');

% COMPUTING ALPHA COEFFICIENTS FOR CLASS-BALANCING
[alpha] = balances(input, label, info);
fprintf('The data was class-balanced.\n');

% RE-COMPUTING DATA INFO
info = data_info(input, label);     %Extracts/displays training dataset information

%% STEP 3: ASSIGNING PARAMETERS
parameters{1} = input;      %Pass on pre-processed training set
parameters{2} = param_z;    %Parameters for z-scoring
parameters{3} = alpha;      %Alpha coefficients for class-balancing
parameters{4} = info;       %Pass on data information
parameters{5} = label;      %Pass on training dataset labels
end




%% THIS FUNCTION EXTRACTS BASIC INFORMATION ON A DATASET
function [info] = data_info(inputs, outputs)
[num_datapts, num_features] = size(inputs);     %Number of datapoints and number of features
class_names = unique(outputs)';                 %Name of classes
num_class = length(class_names);                %Number of classes

%% DISPLAY DATA INFO
fprintf('The complete dataset contains %d data points with %d features.\n',
num_datapts, num_features);
class_string = sprintf('%d ', class_names);
```

```
fprintf('There are %d different classes observed in the training dataset, which can be: %s\n\n', num_class, class_string);

info = {num_datapts, num_features, num_class, class_names};
end


%% THIS FUNCTION Z-SCORES DATA, STORE PARAMETERS MEAN AND STD
function [feat_z, param_z] = standardize(inputs)
[feat_z, mu, sigma] = zscore(inputs);
param_z = [mu; sigma];
end

%% THIS FUNCTION COMPUTES COEFFICIENTS ALPHA REFLECTING CLASS IMBALANCE
function [alpha] = balances(inputs, outputs, info)
%% EXTRACT INFORMATION ON DATASET
class_names = info{4};                                  %Class names (1,2,3,4,5)
num_class = info{3};                                    %Number of classes (5)

%% COUNT NUMBER OF ELEMENTS PER CLASS
count_class = zeros(1,num_class);                       %Initialize vector of number of datapoints per class
for i = 1:num_class                                     %Iterate over classes
    temp = inputs((outputs == class_names(i)),:);       %Select all training datapoints from that class
    count_class(i) = length(temp);                      %Count number of elements per class
end

%% COMPUTE ALPHAS
alpha = zeros(1, num_class);                            %Initialize vector of alpha coefficient per class
maxElements = max(count_class);                         %Compute number of elements in most represented class
for i = 1:num_class                                     %Iterate over classes
    alpha(i) = maxElements/count_class(i);              %Ratio of number of elements in most represented class to number of elements in that class
end
end
```

```
%% CLASSIFIER GCA (generative classification approach, using trained parameters)
%%% Define 1 class probability distribution model per class (Gaussian model assumed)
%%% Compute posterior probability of point "input" to belong to class Ck

%%% Classification rule: classify "input" with class with largest posterior priority

%%% IN: "input" is a row vector containing features of one data point
%%% IN: "parameters" is an array containing trained parameters

function [class] = ClassifyGCA(input, parameters)
%% STEP 0: REMOVE NON-INFORMATIVE FEATURES
input = input(:,1:60);              %Remove features 61 to 64

%% EXTRACT PARAMETERS
means = parameters{1};              %MEANS
inv_cov_class = parameters{2};      %INVERSE COVARIANCE (PER CLASS)
class_names = parameters{3};        %CLASS NAMES
p_class = parameters{4};            %PRIOR CLASS PROBABILITIES
inv_cov_global = parameters{5};     %INVERSE COVARIANCE (GLOBAL)
standard = parameters{6};           %SHOULD DATA BE Z-SCORED
param_z = parameters{7};            %PARAMETERS FOR Z-SCORE

%% SELECT COVARIANCE
cov = 0;
if(cov == 1)
    inv_cov = inv_cov_global;
    fprintf('The global covariance will be used.\n');
else
    inv_cov = inv_cov_class;
    fprintf('The covariance per class will be used.\n');
end
clear cov;

%% EVALUATE EACH CLASS-SPECIFIC GAUSSIAN AT TEST DATAPOINT
[num_class, num_features] = size(means);

for test = 1:length(input)  %Iterate over test datapoints
    %% Z-SCORING
    if(standard == 1)
        input(test,:) = (input(test,:) - param_z(1,:))./(param_z(2,:));
    end

    %% COMPUTE PRIOR P(input|Ck)
    px_given_class = zeros(1,num_class);
    for c = 1:num_class     %Iterate over classes
        px_given_class(c) = exp((-1/2)*(input(test,:)-means(c,:))*inv_cov(:,:,c)*(input(test,:)-means(c,:))');   %Gaussian model assumption
        % NOTE : the exponential agressively pushes large magnitude number to 0 or +infinity
        % Also p(x|Ck) is interpreted for one class with respect to others and exponential is monotonic
```

```
    end

    %% NORMALISATION
    px_given_class = px_given_class/sum(px_given_class);

    %% POSTERIOR PROBABILITY
    % We ignore marginal p(x) since it will be the same across classes
    pclass_given_x = px_given_class.*p_class;

    % Class which maximizes posterior probability
    [max_p, idx] = max(pclass_given_x);     %Assign class according to highest posterior probability of belonging to that class
    DoB = max_p/sum(pclass_given_x);        %Normalised degree of belief
    class(test) = class_names(idx);
    fprintf('This datapoint was classified with %.2f percent certainty as belonging to class %d.\n', DoB*100, class(test));
end
class = class';
end
```

```
%% CLASSIFIER MLP (multi-layer perceptron)
%%% 1. Forward-propagate test datapoint through the network using trained weights and biases
%%% 2. The softmax function is applied to the network output
%%% 3. The neuron with the highest probability of datapoint belonging to its
%%% corresponding class is selected to assign class membership of test datapoint

%%% IN: "inputs" is a list of test datapoints (vectors of values among features)
%%% IN: "parameters" is an array containing parameters trained using TrainsMLP()

function [class, p_max] = ClassifyMLP(inputs, parameters)
%% STEP 0: REMOVE NON-INFORMATIVE FEATURES
inputs = inputs(:,1:60);        %Remove features 61 to 64

%% Extract and assign trained parameters
W = parameters{1};              %Network weights
NameOfClasses = parameters{2};  %Vector containing names of classes
MeanAndStdZ = parameters{3};    %Mean and standard deviation of training dataset used to z-score the test dataset
standardize = parameters{4};    %Boolean control indicating whether test data should be z-scored
nbrOfLayers = parameters{5};    %Number of network layers
beta = parameters{6};           %Activation gain beta
B = parameters{7};              %Networks biases

%% Z-SCORE TEST DATASET (PRE-PROCESSING)
for i = 1:length(inputs)    %Iterate over the full test dataset
    %% Z-SCORE
    if(standardize == 1)
        inputs(i,:) = (inputs(i,:) - MeanAndStdZ(1,:))./(MeanAndStdZ(2,:));
    end

    %% FORWARD PASS
    LayerInput{1} = inputs(i,:)';   %Layer input is one test datapoint
    for layer = 1:(nbrOfLayers-1)   %Iterate over network layers
        input = LayerInput{layer};  %Assign input to that layer
        weights = W{layer};         %Assign weight connecting that layer to next
        u{layer+1} = weights*input + B{layer};                  %Compute layer activity
        LayerOutput{layer+1} = activation(u{layer+1}, beta);    %Compute layer activation
        LayerInput{layer+1} = LayerOutput{layer+1};             %Output of one layer is input to the next
    end

    %% CLASS ASSIGNMENT
    output = exp(u{nbrOfLayers})./sum(exp(u{nbrOfLayers}));     %Softmax operator (convert data over range [-1:+1] to range [0:1])
    [p_max(i), class(i)] = max(output);                         %Neuron with highest activation assigns class membership of test datapoint
end
class = class';
end
```

```
%% NEURON ACTIVATION FUNCTION
function [output] = activation(input, beta)
output = tanh((beta/2)*input);
end
```

```
%% CLASSIFIER KNN (k-nearest-neighbour)
%%% 1. Extract parameters trained through TrainClassifierX
%%% 2. Pre-process the test data through z-scoring with trained parameters
%%% 3. Identify k training datapoints that are closest to test data point
%%% 4. Vote of each neighbour is weighted by inverse square distance to the test datapoint
%%% 5. Assign class according to voting majority class membership among the k neighbours
%%% NOTE: We use euclidian distance as metric

%%% IN: "input" is a matrix containing test datapoint as rows
%%% IN: "parameters" is a set of trained parameters from TrainClassifierX
%%% OUT: "label" is a vector containing the predicted class of each test datapoint

function [label] = ClassifyX(input, parameters)
%% STEP 0: REMOVE NON-INFORMATIVE FEATURES
input = input(:,1:60);              %Remove features 61 to 64

%% STEP 1: EXTRACT PARAMETERS
training_set = parameters{1};       %Pre-processed training set
param_z = parameters{2};            %Parameters for z-scoring test datapoints
alpha = parameters{3};              %Alpha coefficients for class-balancing
info = parameters{4};               %Basic information on training dataset
training_labels = parameters{5};    %Labels for the training set

%SET HYPERPARAMETER K
K = 5;

%Extract basic informations about training dataset
nbrOfTrainDatapoints = info{1};     %Number of training examples
nbrOfFeatures = info{2};            %Size of feature space
nbrOfClasses = info{3};             %Number of possible classes
ClassNames = info{4};               %Names of classes

%% STEP 2: CLASSIFIER
for test = 1:length(input(:,1))     %Iterate over test datapoints
    %% STEP 2.1: PREPROCESSING (z-score test datapoint with parameters learnt on training set)
    input(test,:) = (input(test,:) - param_z(1,:))./(param_z(2,:));

    %% STEP 2.2: COMPUTE DISTANCE OF TEST DATAPOINT TO ALL TRAINING DATAPOINTS
    distance = zeros(1,nbrOfTrainDatapoints);                       %Initialize vector of distances of test datapoint to all training datapoints
    for i = 1:nbrOfTrainDatapoints                                  %Iterate over training datapoints
        distance(i) = norm(input(test,:) - training_set(i,:));      %Euclidian distance
    end
    [dist_order, idx] = sort(distance, 'ascend');                   %Organize distances in ascending order

    %% STEP 2.3: IDENTIFY CLASS OF K-NEAREST NEIGHBOURS
    class_neighbours = zeros(2,K);                              %Initialize vector of
```

```
class name and vote weight (first and second rows) for each neighbour (columns)
    for j = 1:K                                                 %Iterate over↙
neighbours
        class_neighbours(1,j) = training_labels(idx(j));     %Store class of↙
neighbour
        %Store weight of neighbour vote including alpha and distance criteria
        class_neighbours(2,j) = alpha(class_neighbours(1,j))/dist_order(j);
    end

    %% STEP 2.4: COMPUTE VOTE PER CLASS
    NeighboursClasses = unique(class_neighbours(1,:));  %Classes expressed amongst↙
neighbours
    vote = zeros(2,length(NeighboursClasses));          %Intialize vote per class

    for elem = 1:length(NeighboursClasses)              %Iterate over classes↙
represented amonst neigbours
        vote(1,elem) = NeighboursClasses(elem);         %Accumulate weighted votes↙
for each class
        for i = 1:length(class_neighbours)              %Iterate among K-nearest-↙
neighbours
            if(class_neighbours(1,i) == NeighboursClasses(elem))
                %Aggregate vote for each class among K-nearest neighbours
                vote(2,elem) = vote(2,elem) + class_neighbours(2,elem);
            end
        end
    end

    %% ASSIGN CLASS BASED ON MAJORITY VOTING
    [val, idx_max] = max(vote(2,:)); clear val;
    label(test) = vote(1, idx_max);
end
label = label';
end
```