MACHINE LEARNING AND PATTERN RECOGNITION
Assignment 1

Matriculation number - s1569105
Examination number - B076165

November 2015

1 The Next Pixel Prediction Task

Note

For all code snippets in the part 1 it is assumed that I have loaded imgregdata.mat file via matlab terminal before I ran the scripts. For the tasks starting from 1.3 it is also assumed that I have loaded NetLab and welltrainedMLP.mat file.
I don't calculate RMSE upper and lower bounds for cross-validation because it is quite difficult to achieve this with default matlab crossval function. But in such cases I report RMSE with bounds on the test set.

1.1 Data preprocessing and visualization

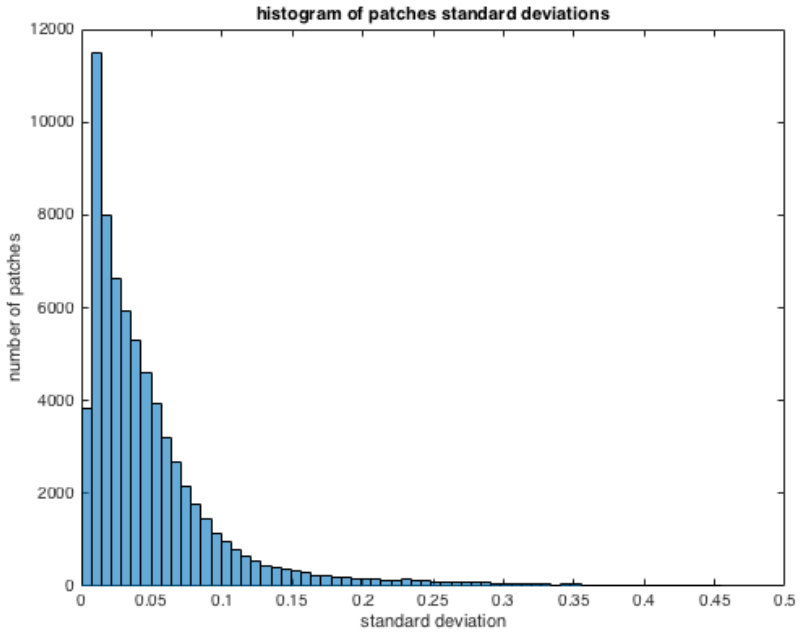


Figure 1: histogram of standard deviations in the xtr dataset after normalisation

(a) The maximum possible value of standard deviation is $\frac{max.-min.}{2}$, so in our case after normalisation it is $\frac{1-0}{2} = 0.5$. Our threshold to distinguish discrete values of pixels is $\frac{1}{64} = \approx 0.0156$. If we use 32 bins, for example, on a range of possible values of standard deviation (between 0 and 0.5) then the width of one bin will be 0.0156 and standard deviations with values 0.0151 or 0.0021 will go to the same bin. But we usually would associate standard deviation 0.0151 with the discrete (original) pixel value of 1 and 0.0021 with 0 because

$round(0.0151/0.0156 - 1) = round(0.968) = 1$ and $round(0.0021/0.0156) = round(0.135) = 0$. Therefore, we must choose minimum 64 bins in order to distinguish such cases because we will have bins width $\frac{0.5}{64} \approx \frac{0.0156}{2} = 0.0078$ and each bin will correspond to the specific discrete (original) pixel value.
From the figure 1 we can see that after the peak on the second bin (associate it with deviation around 1 discrete pixel value) the number of patches declines exponentially as standard deviation increases . We can conclude that most of patches have standard deviation between 0 and 0.05. It can be confirmed from subtask c. 52739 patches which is more than 75% of our initial dataset have standard deviation within 0 and 0.0635 range, and 0.0635 is quite small standard deviation. Therefore, most of the patches are flat ones.
Code snippet to plot histogram:

```
%launch via - tsk1_1_a(xtr)
function [] = tsk1_1_a(xtr)
    patches = xtr ./ 63;
    patches_std = std(patches,0,2);

    figure;
    histogram(patches_std,64);
    title('histogram of patches standard deviations');
    xlabel('standard deviation');
    ylabel('number of patches');
end
```

(b) I would choose the mean of all pixels in the patch as simple predictor for flat patches. Given the definition of flat patches the intensity of the i'th pixel in the flat patch should be something like this $f_i(\boldsymbol{x_{all\ other\ pixels}}) = const_{this flat\ patch} + o(\boldsymbol{x_{all\ other\ pixels}})$ where $o(\boldsymbol{x_{all\ other\ pixels}})$ is small function in comparison to $const_{flat\ patch}$, and $o(\boldsymbol{x_{all\ other\ pixels}})$ mean and standard deviation over all pixels in the patch are 0 and $\sigma_{flat\ patch}$ respectively. For flat patches the following is true $\sigma_{flat\,patch} \leq \sigma_{flat\,pach\,max}$. So it is natural to propose $const_{this flat\ patch}$ as our prediction, which can be received by calculating mean over all pixels in the flat patch
The performance of this simple predictor can be estimated by considering extreme case when after normalisation (all pixel values between 0 and 1) most pixels are zeroes and small portion of pixels are ones (correspond to original intensity of 63). Let $N - m$ be number of zeros and let m be number of ones and I denote μ as mean.

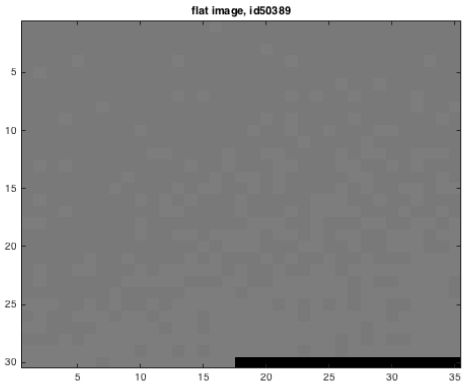
$$
\begin{aligned}
m &< N - m \\
\mu &= \frac{(N-m)*0 + m*1}{N} = \frac{m}{N} \\
\sigma^2 &= \frac{1}{N}[(N-m)(0-\frac{m}{N})^2 + m(1-\frac{m}{N})^2] \\
&= \frac{(N-m)m^2}{N^3} + \frac{m(N-m)^2}{N^3} \\
N^3\sigma^2 &= m^2N - m^3 + mN^2 - 2m^2N + m^3 \\
&= mN^2 - m^2N \\
m^2 - mN + N^2\sigma^2 &= 0 \\
m = \frac{N}{2}(1 - \sqrt{1-4\sigma^2}) &\quad \text{(minus because our case is } m < N - m \text{)}
\end{aligned}
$$

putting $\sigma_{flat\,pach\,max} = \frac{4}{63} \approx 0.0635$ instead of σ and using $N = 1032$ we get

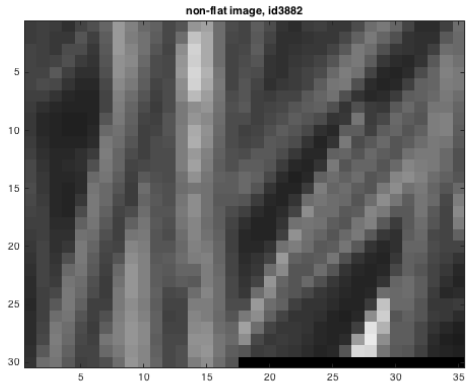
$$m = \frac{1032}{2}(1 - \sqrt{1 - 4 * 0.0635^2}) \approx 4.178$$

rounding m to the closest integer we receive $m = 4$.
Thus, in most extreme case of flat patch we can have 4 ones (correspond to original pixel intensity of 63) and 1028 zeros, so it is natural that we want to predict zero as discrete value of our target pixel. The mean gives us $\mu = \frac{1028*0+1*4}{1032} \approx 0.0038$. Dividing range between 0 and 1 by 64 we get 0.0156 as our threshold to distinguish discrete pixel values. 0.0038 lies between 0 and 0.0156 so our mean value would correspond to 0 as the discrete value of our target pixel and that is what we wanted.

Figure 2: patch images



(a) flat patch image



(b) non-flat patch image

(c) Code snippet to show patch images on figure 2:

```
%launch via tsk1_1_c(xtr)
function [] = tsk1_1_c(xtr)
    %normalising
    patches = xtr ./ 63;
    patches_std = std(patches,0,2);

    %the threshold 4/63 is taken from the task
    flat_threshold = 4/63;
    flat_patches_ids     = patches_std <= flat_threshold;
    non_flat_patches_ids = patches_std >  flat_threshold;

    %split on flat and non-flat patches
    flat_patches     = patches(flat_patches_ids, :);
    non_flat_patches = patches(non_flat_patches_ids,:);

    function [rnd_image, rnd_image_id] = get_rnd_image(patches)
        rnd_image_id = randi(size(patches, 1), 1);
        patch = patches(rnd_image_id, :);
        %expanding patch to the full size
        patch(1050) = 0;
        %reshaping to image
        rnd_image = reshape(patch, [35, 30]);
        %transposing them to ensure right position on the plot
        rnd_image = rnd_image';
    end

    function [] = show_image(image, title_str, image_id)
        figure;
        imagesc(image, [0, 1]);
        title(strcat(title_str, num2str(image_id)));
        colormap gray;
    end

    [flat_image, flat_id] = get_rnd_image(flat_patches);
    [non_flat_image, non_flat_id] = get_rnd_image(non_flat_patches);

    show_image(flat_image, 'flat image, id ', flat_id);
    show_image(non_flat_image, 'non-flat image, id ', non_flat_id);
end
```

1.2 Linear regression with adjacent pixels

(a) I used 5000 training points from xtr_nf and ytr_nf to plot figure 3. From it we can see that x(j, end), x(j, end - 34), y(j) are strongly positively correlated. However, there is some relatively small number of deviations from this trend. It seems that these deviations are normally distributed so linear regression should be reasonable model to describe such data. Code snippet for scatter plot:

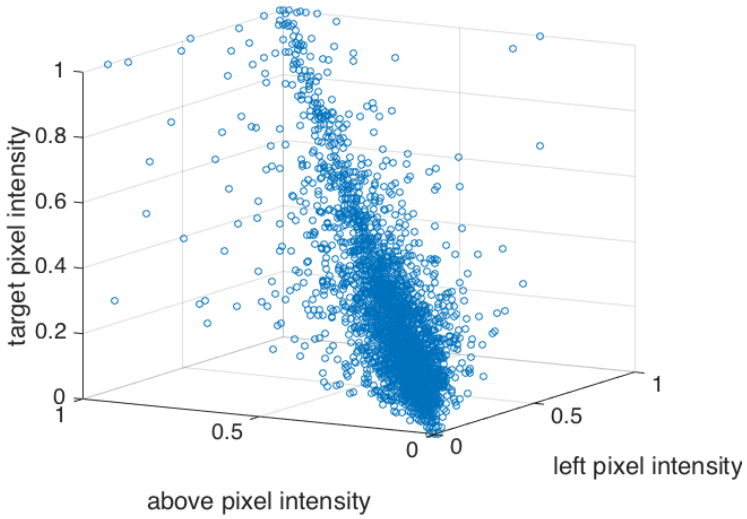


Figure 3: scatter plot of adjacent pixels. 5000 data points from xtr_nf and ytr_nf

```
%launch via - tsk1_2_a(xtr_nf, ytr_nf)
function [] = tsk1_2_a(xtr_nf, ytr_nf)
    %I choose 5000 so plot doesn't look cluttered
    num_data_points = 5000;
    x_left   = xtr_nf(1:num_data_points, end);
    x_above  = xtr_nf(1:num_data_points, end - 34);
    x_target = ytr_nf(1:num_data_points);

    scatter3(x_left, x_above, x_target);
    xlabel('left pixel intensity');
    ylabel('above pixel intensity');
    zlabel('target pixel intensity');
    set(gca,'FontSize', 20);
end
```

(b) Derivation of this solution can be taken from MLPR lecture 7 slides 8-11 here. The solution for weights from there is:

$$\hat{\boldsymbol{w}} = (\Phi^T\Phi)^{-1}\Phi^T\boldsymbol{y}$$

In our notation matrix Φ will become:

$$\Phi = X = \begin{pmatrix} 1, x(1, end), x(1, end-34) \\ 1, x(2, end), x(2, end-34) \\ \dots \\ 1, x(N, end), x(N, end-34) \end{pmatrix}$$

$$\hat{\boldsymbol{w}} = (X^TX)^{-1}X^T\boldsymbol{y}$$

where N is a number of training data points and x is our dataset (it will be xtr_nf in the next task)

(c) After training the weights are:

bias	left pixel	above pixel
0.0026	0.4606	0.5241

the RMSE for test and training sets:

	Training set	Test set
RMSE	0.0506 ± 0.0010	0.0503 ± 0.0017

Taking bounds into consideration we can see that the performance on training and test is almost the same. That is why we can conclude that linear regression is not over-fitting the data in this problem. It can be seen from figure 4 that indeed there is strong positive correlation between adjacent pixels and target value pixel.
code snippet to get linear regression predictor (will also be used afterwards):

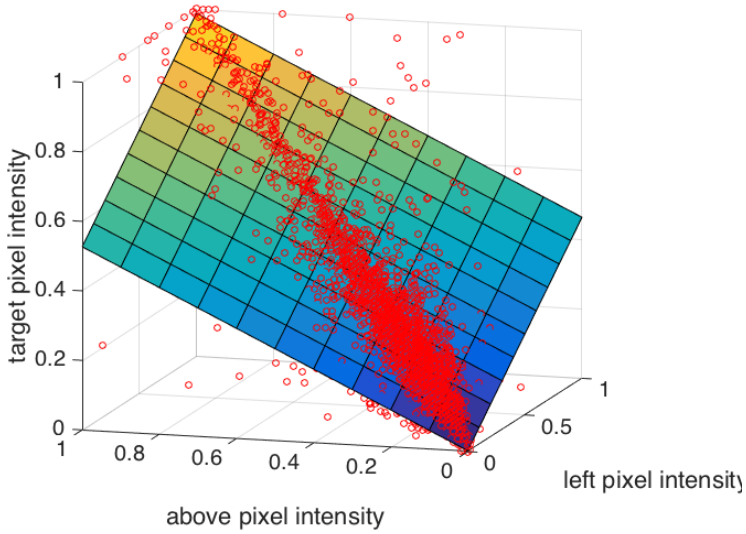


Figure 4: plot of the linear regression function after training along with test data points

```
function [w, predictor] = cs_linear_regression(x_train, t_train)
    %custom linear regression
    %it inserts bias term automatically %Ruslan Burakov

    %adding bias term
    calc_Phi = @(x)[ones(size(x, 1), 1), x];

    %computing weights
    w = pinv(calc_Phi(x_train)) * t_train;

    predictor = @(x_test)calc_Phi(x_test) * w;
end
```

code snippet to compute root mean square error (RMSE) with bounds, which correspond to standard errors of mean square error(MSE) (will also be used afterwards):

```
function [rmse_est, std_err] = cs_rmse(t, y)
    %custom root mean square error
    diff = t - y;
    %mean square error
    sqr_diff = diff .* diff;
    mse_est = mean(sqr_diff);
    rmse_est = sqrt(mse_est);
    if nargout > 1
        %professor Chris Williams pointed out that
        %to produce some bounds on rmse:
        %mse = mse_est +/- mse_std_err
        %rmse = f(mse) = f(mse_est +/- mse_std_err) = sqrt(mse_est +/- mse_std_err)
        %f(u + h) ~= f(u) + f'(x)h = sqrt(u) + 0.5 * h/(sqrt(u)) %Taylor expansion
        S = numel(t);
        var_est = var(sqr_diff);
        mse_std_err = sqrt(var_est/S);
        std_err = 0.5 * mse_std_err / rmse_est;
    end
end
```

code snippet to show rmse on training and test set with bounds (will also be used afterwards):

```
function [] = show_rmse(t_train, y_train, t_test, y_test)
    [rmse_train, std_err_train] = cs_rmse(t_train, y_train);
    [rmse_test,  std_err_test]  = cs_rmse(t_test,  y_test);

    fprintf('rmse on training set %5.4f +/- %5.4f\n', rmse_train, std_err_train);
    fprintf('rmse on test set %5.4f +/- %5.4f\n', rmse_test, std_err_test);
end
```

code snippet for this task:

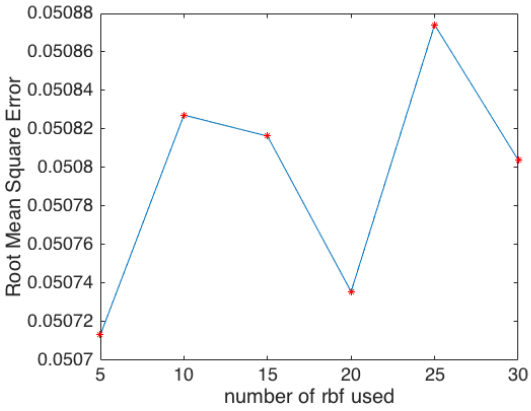
```
%launch via - tsk1_2_c(xtr_nf, ytr_nf, xte_nf, yte_nf)
function [] = tsk1_2_c(x_all_train, t_train, x_all_test, t_test)
    %t - means target values
    get_adjacent_pixels = @(x) [x(:, end), x(:, end - 34)];

    %prepare sets
    x_train = get_adjacent_pixels(x_all_train);
    x_test  = get_adjacent_pixels(x_all_test);

    %train (it will add bias term automatically)
    [w, predictor] = cs_linear_regression(x_train, t_train);
    display(w, 'weights for neighbours pixels features');

    show_rmse(t_train, predictor(x_train), t_test, predictor(x_test));

    %show surface
    figure,
    %I chose smaller step because our function is just a plane
    [dim1, dim2] = meshgrid(0:0.1:1, 0:0.1:1);
    %swapped ones from original snippet, because w(1) corresponds to bias
    %in my case
    ysurf = [ones(numel(dim1),1), [dim1(:), dim2(:)]] * w;
    surf(dim1, dim2, reshape(ysurf, size(dim1)));

    hold on;

    %show test set data points
    scatter3(x_test(:, 1), x_test(:, 2), t_test, 'red');
    xlabel('left pixel intensity');
    ylabel('above pixel intensity');
    zlabel('target pixel intensity');
    set(gca,'FontSize', 20);
end
```

1.3 RBF regression with adjacent pixels

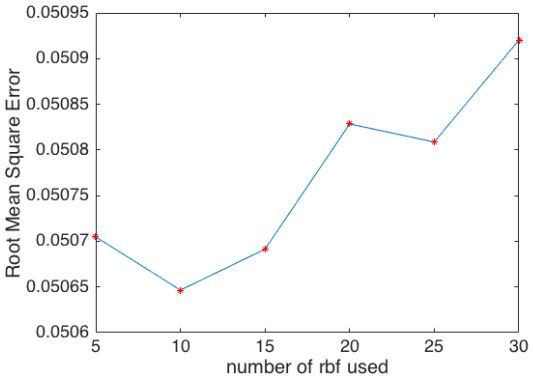
(a) When I ran cross validation procedure to determine which number of radial bases functions among { 5, 10, 15, 20, 25, 30} produces the best results, each time I received a different answer. The figure 5a suggests 5 as the best number of radial bases functions and 5b proposes 10 as the best choice. This happens probably due to the random numbers as matlab crossval uses them each time to divide input set on training and validation sets and rbf network initialises weights differently depending on the random numbers. After that I launched the procedure for number of radial basis functions between 1 and 20, and I have realised that it was just a matter of scale. The figure 5c demonstrates that we achieve almost no improvement if we use more than 5 radial bases functions in this task. That is why I have chosen 5 as my number of radial basis functions because for the same efficiency it takes less time to compute. Code snippet for this task:

```
%launch via
%tsk1_3_a(xtr_nf, ytr_nf, 5:5:30)
%and
%tsk1_3_a(xtr_nf, ytr_nf, 1:20)
function [] = tsk1_3_a(x_all, t, num_rbfs)
    %t - means target values
    get_adjacent_pixels = @(x) [x(:, end), x(:, end - 34)];

    x = get_adjacent_pixels(x_all);

    opt = foptions;
    opt(1)  = 1; % Display EM training
    opt(14) = 5; % number of iterations of EM
    dim     = 2; % left_pixel, above_pixel in our case

    function regf = create_rbf_regf(num_rbf)
        function y_test = rbf_reg(x_train, t_train, x_test)
            net = rbf(dim, num_rbf, 1, 'gaussian');
            net = rbftrain(net, opt, x_train, t_train);
            y_test = rbffwd(net, x_test);
        end
        regf = @rbf_reg;
    end
```

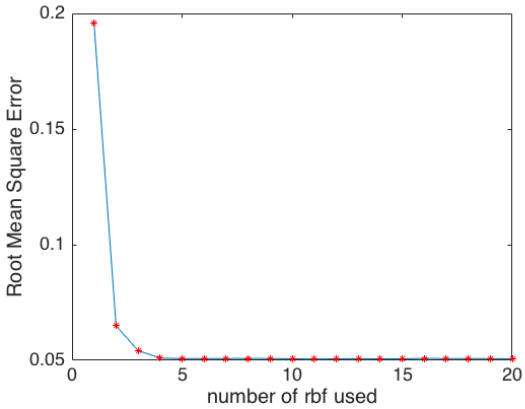
Figure 5: Root Mean Square Error against number of radial basis functions used



(a)



(b)



(c)

```
    rmse_records = zeros(1, length(num_rbfs));

    for i = 1:length(num_rbfs)
        num_rbf = num_rbfs(i);
        regf = create_rbf_regf(num_rbf);
        %default CV 10 folds
        cvMse = crossval('mse', x, t,'predfun',regf);
        rmse_records(i) = sqrt(cvMse);
    end

    plot(num_rbfs, rmse_records)
    hold on;
    plot(num_rbfs, rmse_records, 'r*');
    xlabel('number of rbf used');
    ylabel('Root Mean Square Error');
    set(gca,'FontSize', 18);
end
```

(b) the RMSE for test and training sets:

	Training set	Test set
RMSE	0.0506 ± 0.0010	0.0503 ± 0.0017

As it can be seen that radial basis functions don't give any noticeable improvement in comparison to linear regression when both of them are using only adjacent pixels as features.

Code snippet for this task:

```
%launch via - tsk1_3_b(xtr_nf, ytr_nf, xte_nf, yte_nf)
function [] = tsk1_3_b(x_all_train, t_train, x_all_test, t_test)
    %t - means target values
    get_adjacent_pixels = @(x) [x(:, end), x(:, end - 34)];

    x_train = get_adjacent_pixels(x_all_train);
    x_test  = get_adjacent_pixels(x_all_test);

    opt = foptions;
    opt(1)  = 1; % Display EM training
    opt(14) = 5; % number of iterations of EM
    dim     = 2; % left_pixel, above_pixel in our case
    num_rbf = 5;% determined from previous task

    net = rbf(dim, num_rbf, 1, 'gaussian');
    net = rbftrain(net, opt, x_train, t_train);

    show_rmse(t_train, rbffwd(net, x_train), t_test, rbffwd(net, x_test));
end
```

1.4 Linear regression with all pixels

the RMSE for test and training sets:

	Training set	Test set
RMSE	0.0371 ± 0.0007	0.0456 ± 0.0018

There is an improvement comparing with both previous methods but on the test set it is quite small. Overall, it is possible to conclude that whereas knowledge of all the pixels helps to predict the value of target pixel better, it seems that the most significant features are pixels adjacent to the target one. Code snippet for this task:

```
%launch via - tsk1_4(xtr_nf, ytr_nf, xte_nf, yte_nf)
function [] = tsk1_4(x_train, t_train, x_test, t_test)
    %t - means target values
    %train (it will add bias term automatically)
    [w, predictor] = cs_linear_regression(x_train, t_train);
    show_rmse(t_train, predictor(x_train), t_test, predictor(x_test));
end
```

1.5 Neural Network with all pixels

(a) the RMSE for test and training sets:

	Training set	Test set
RMSE	0.0333 ± 0.0006	0.0473 ± 0.0017

In comparison to linear regression with all pixels Neural Network (NN) slightly over-fits the data because its error is lower on training set but the error on test set is larger. In my opinion, this happens because NN with all pixels is very sophisticated model for our task. Code snippet for this task:

```
%launch via - tsk1_5_a(net, xtr_nf, ytr_nf, xte_nf, yte_nf)
function [] = tsk1_5_a(net, x_train, t_train, x_test, t_test)
    %t - means target values
    show_rmse(t_train, mlpfwd(net, x_train), t_test, mlpfwd(net, x_test));
end
```

(b) Neural Network RMSE for different random seeds:

Random seed	RMSE on full training set	RMSE on test set
2015	0.0500 ± 0.0009	0.0515 ± 0.0014
2016	0.0477 ± 0.0009	0.0504 ± 0.0014
2017	0.0485 ± 0.0009	0.0515 ± 0.0014
2018	0.0477 ± 0.0009	0.0516 ± 0.0014
2019	0.0489 ± 0.0009	0.0527 ± 0.0016

In comparison to previous task 1.5.a the difference between training and test errors is much smaller in all cases because we are using only 5000 instances from training set for learning. At the same time, most likely due to the same reason, our performance on the training and test set has dropped.
Random seed determines to what local minimum NN will try to converge and the starting point from which it will try to converge. So depending on random seeds some runs will converge faster than others. It explains why our results differ but taking bounds into consideration our performance on the test set is roughly the same for all runs.
Code snippet for this task:

```
%launched via - tsk1_5_b(xtr_nf, ytr_nf, xte_nf, yte_nf)
function [] = tsk1_5_b(x_train, t_train, x_test, t_test)
    %t - means target values
    nhid = 10; % number of hidden units

    % Set up vector of options for the optimiser.
    options = zeros(1,18);
    %options(1)  = 1; % This provides display of error values.
    %options(9)  = 1; % Check the gradient calculations.
    options(14) = 200; % Number of training cycles.

    function [] = launch_NN(seed)
        rng(seed,'twister');
        net = mlp(size(x_train,2), nhid, 1, 'linear');
        [net, tmp] = netopt(net, options, x_train(1:5000,:), t_train(1:5000,:), 'scg');

        fprintf('seed %4.0f:\n', seed);
        show_rmse(t_train, mlpfwd(net, x_train), t_test, mlpfwd(net, x_test));
    end

    seeds  = 2015:1:2019;

    for i = 1:length(seeds)
        seed = seeds(i)
        launch_NN(seed);
    end
end
```

1.6 Discussion

In previous tasks linear regression with all pixels showed the best performance. That is why I thought that it would be reasonable to try something else with linear regression. It seemed to me that the adjacent pixels have the biggest impact on the intensity of the target pixel. But how many nearest pixels do we need? I wasn't sure so I evaluated linear regression performance depending on the number of pixels from the end of feature vector via cross-validation on the training set. The result can be seen on the figure 6. **Note**: the plot step is not uniform, after 182 I used much bigger step (100) because otherwise cross-validation would take too much time.
Figure suggests that the best value from the end is something like 120. Using that number of feature on the test set (xte_nf(:, 912:1032)) I received the following RMSE:

	Test set
RMSE	0.0419 ± 0.0017

This result shows improvement in comparison to the previous models. Also we can see that at the beginning figure 6 has form similar to the step function. I describe it by the fact that with period of 35 we receive pixels which are above the target one and they give a lot of information about target pixels. This leads to conclusion that if image is 2d-array of pixels where i, j are indexes on the X and Y axis respectively then we should select pixels/features depending on their euclidean distance from the target pixel $r = \sqrt{(18-i)^2 + (30-j)^2}$.
I have performed cross-validation on the training set to find the best such distance r, the results are on figure 7. It points out that the best distance is 4, and after checking it on the test set:

	Test set
RMSE	0.0412 ± 0.0017

This is good result and this model is computationally efficient because it needs to take only 21 closest pixels in contrast to linear regression or Neural Network with all the pixels.
Code snippet to plot figure 6:

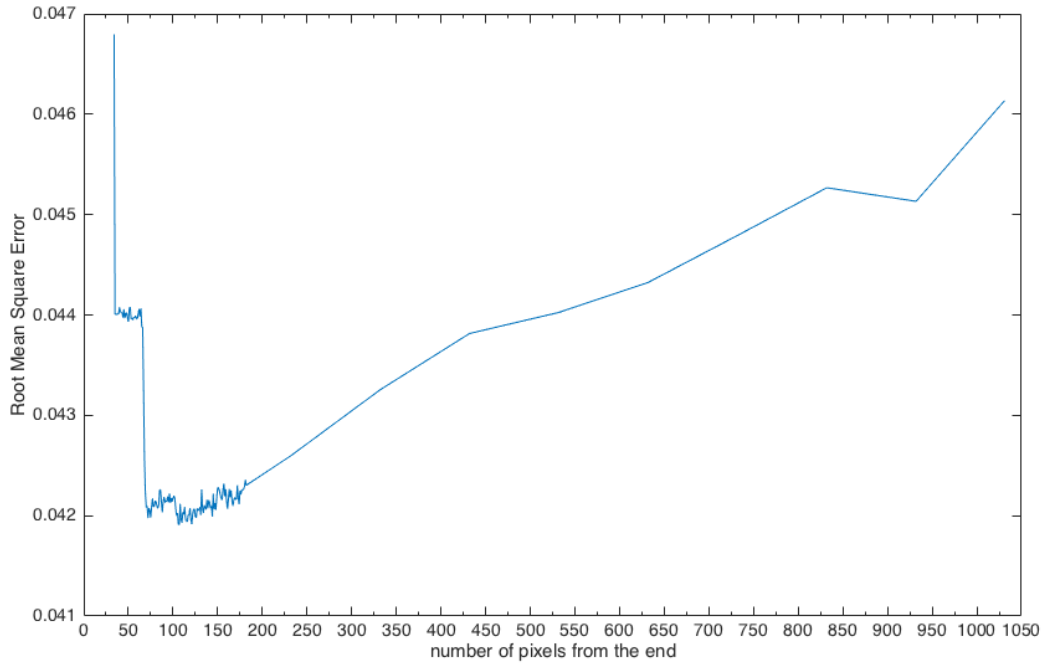


Figure 6: plot of linear regression RMSE depending on the number of pixels from the end used as features.

```
%launch via - tsk1_6_1(xtr_nf, ytr_nf)
function [] = tsk1_6_1(x_all_train, t)
    %t - means target values
    function y_test = regf(x_train, t_train, x_test)
       [w, predictor] = cs_linear_regression(x_train, t_train);
       y_test = predictor(x_test);
    end

    offsets = (1032 - 34):-1:850;
    offsets = [offsets, 800, 700, 600, 500, 400, 300, 200, 100, 1];
    errors = zeros(length(offsets), 1);

    for i = 1:length(offsets)
        offset = offsets(i)
        x = x_all_train(:, offset:1032);
        %default 10 fold
        cvMse = crossval('mse', x, t,'predfun', @regf);
        errors(i) = cvMse ^ 0.5;
    end

    pixels_from_end = 1032 - offsets;
    plot(pixels_from_end, errors);
    xlabel('number of pixels from the end');
    ylabel('Root Mean Square Error');
end
```

Code snippet to get performance on the test set using 120 pixels from the end:

```
%launch via - tsk1_6_2(xtr_nf, ytr_nf, xte_nf, yte_nf)
function [] = tsk1_6_2(x_all_train, t_train, x_all_test, t_test)
    %t - means target values
    x_train = x_all_train(:, 912:end);
    x_test  = x_all_test (:, 912:end);

    [w, predictor] = cs_linear_regression(x_train, t_train);

    [rmse_test,  std_err_test]  = cs_rmse(t_test, predictor(x_test));
    fprintf('rmse on test set %5.4f +/- %5.4f\n', rmse_test, std_err_test);
end
```

Code snippet to get the closest pixels:

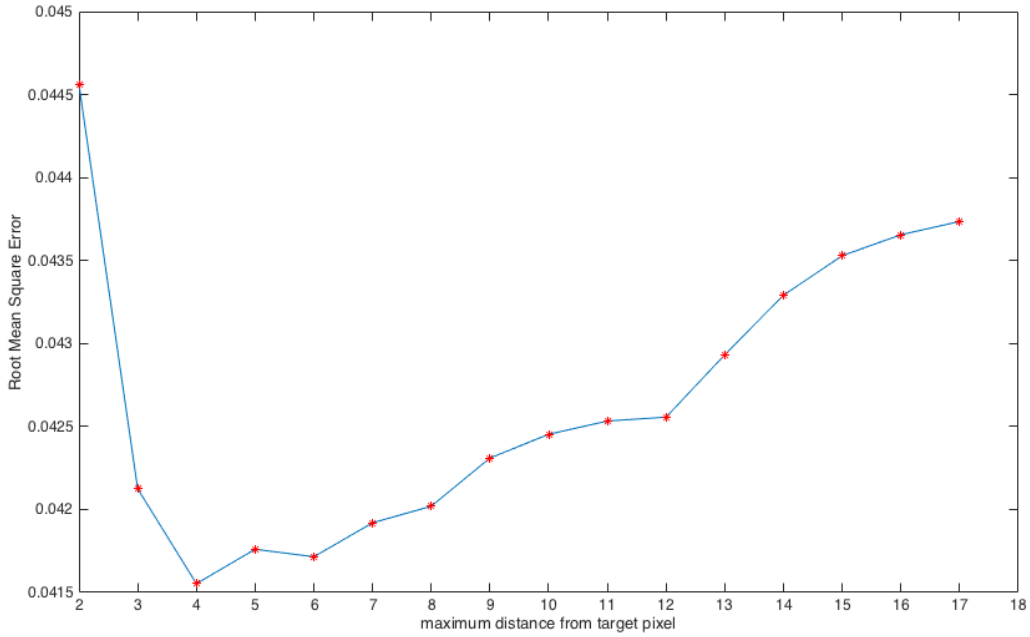


Figure 7: plot of linear regression RMSE depending on the maximum distances from the target pixels within which pixels were used as features.

```
function x = get_closest_pixels(x_all, maximum_distance)
    function accepted = filter(vector_index)
        i = mod(vector_index, 35);
        j = vector_index / 35;
        r = sqrt((18 - i)^2 + (30 - j)^2);
        accepted = r < maximum_distance;
    end

    indexes = arrayfun(@filter, 1:1032);
    x = x_all(:, indexes);
end
```

Code snippet to plot figure 7:

```
%launch via - tsk1_6_3(xtr_nf, ytr_nf)
function [] = tsk1_6_3(x_all_train, t)
    %t - means target values

    function y_test = regf(x_train, t_train, x_test)
       [w, predictor] = cs_linear_regression(x_train, t_train);
       y_test = predictor(x_test);
    end

    max_distances = 2:17;
    rmse_records = zeros(length(max_distances), 1);

    for i = 1:length(max_distances)
        max_dist = max_distances(i)
        x = get_closest_pixels(x_all_train, max_dist);
        %to see complexity of the model
        size(x)
        %default 10 fold
        cvMse = crossval('mse', x, t,'predfun', @regf);
        rmse_records(i) = cvMse ^ 0.5;
    end

    plot(max_distances, rmse_records);
    hold on;
    plot(max_distances, rmse_records, 'r*');
    xlabel('maximum distance from target pixel');
    ylabel('Root Mean Square Error');
end
```

Code snippet to get performance on the test set with pixels within radius 4 from target one:

```
%launch via - tsk1_6_4(xtr_nf, ytr_nf, xte_nf, yte_nf)
function [] = tsk1_6_4(x_all_train, t_train, x_all_test, t_test)
    %t - means target values
    x_train = get_closest_pixels(x_all_train, 4);
    x_test  = get_closest_pixels(x_all_test, 4);

    [w, predictor] = cs_linear_regression(x_train, t_train);

    [rmse_test,  std_err_test]  = cs_rmse(t_test, predictor(x_test));
    fprintf('rmse on test set %5.4f +/- %5.4f\n', rmse_test, std_err_test);
end
```

2 Robust modelling

Note

It is assumed that I have loaded the code from part2_code_data.tar.gz in my matlab environment. I have also taken professor Iain Murray code from MLPR tutorial 5 to produce error bars - errobar_str.m. text_data.mat is assumed to be loaded as well.

2.1 Fitting the baseline model

(a) **Bias feature**
The code snippet to add bias term:

```
%adding bias term
add_bias = @(x)[x, ones(size(x, 1), 1)];

x_train = add_bias(x_train);
x_test  = add_bias(x_test);
```

(b) **Maximizing the likelihood**
The code snippet to report logistic regression performance with error bars:

```
function [] = report_lr(ww, xx, yy, type_str)
    %report logistic regression
    sigmas = 1./(1 + exp(-yy.*(xx*ww))); %logistic regression function
    accuracy = sigmas > 0.5;
    log_sigmas = log(sigmas);

    fprintf('%s meadian of log prob = %5.4f\n', type_str, median(log_sigmas));
    fprintf('%s standard deviation of log prob = %5.4f\n', type_str, std(log_sigmas));
    fprintf('%s accuracy = %s \n',type_str ,errorbar_str(accuracy));
    fprintf('%s log probability = %s \n',type_str, errorbar_str(log_sigmas));
end
```

The code snippet for this and next task:

```
%launch via:
% b) tsk2_1_bc(x_train, y_train, x_test, y_test)
% c) tsk2_1_bc(x_train, y_train, x_test, y_test, 1:100)
function [] = tsk2_1_bc(x_train, y_train, x_test, y_test, varargin)
    MAX_LIN_SEARCHES = 8000;
    if length(varargin) == 0
        train_limits = 1:size(x_train, 1);
    else
        train_limits = varargin{1};
    end

    function [Lp, dLp_dw] = target_fun(ww, xx, yy)
        [Lp, dLp_dw] = lr_loglike(ww, xx, yy);
        Lp = -1 * Lp;
        dLp_dw = -1 * dLp_dw;
    end

    function ww = train(xx, yy)
        initial_ww = ones(size(xx, 2), 1);
        ww = minimize(initial_ww, @target_fun, MAX_LIN_SEARCHES, xx, yy);
    end

    weights = train(x_train(train_limits, :), y_train(train_limits, :));
    report_lr(weights, x_train(train_limits, :), y_train(train_limits, :), 'training set');
    report_lr(weights, x_test,  y_test,  'test set');
end
```

I set initial weights to zeros as my starting point in minimisation procedure, because in that case at the starting point linear regression is unbiased and will give equal probabilities to both labels for any data point. In other words, it will start as a baseline predictor $P(y|\boldsymbol{x}) = 0.5$.
I have received the following results with standard errors:

	Training set	Test set
Accuracy	0.8335 ± 0.0046	0.9071 ± 0.0072
Mean log probability	-0.4398 ± 0.0081	-0.300 ± 0.012
Variance of log probability	0.4292	0.2234
Median of log probability	-0.2339	-0.1998

Not a typo, my accuracy was better on the test set. Probably this indicates that training set is noisy (has some mislabelled records) but logistic regression is robust enough to capture general pattern instead of fitting the noise. And if we suggest that the test set was created more carefully then it will explain better accuracy on the test set.
The minimisation function converged by itself (I put very big value for maximum number of linear searches - 8000) on 4921 linear search. The log probability of the baseline line for each input is $log(P(y|\boldsymbol{x})) = log(0.5) = -0.6931$. The closer $mean(log(P(y|\boldsymbol{x})))$ to zero the more confident predictor in the right answers. Although we must remember the fact that we can have good predictor with good accuracy, but if it gives chances which is close to zero to the right answer even to the one data point then due to the nature of log function ($log(01) = -\infty$) our log mean quantity will be skewed heavily and much smaller than zero. So theoretically it is possible to have predictor with 99.99% accuracy but with mean log probability far below zero. That is why I also report variance and median of log probability. These quantities will be used for comparison in the next tasks.
As we can see our mean log probability in test set is closer to 0 than those of simple baseline which together with our accuracy implies that our performance is better than performance of baseline.

(c) **Limited training data**
As in the previous subtask I used zeros as my initial weights.
For limited number of training examples, my results are:

	The first 100 instances from training set	Test set
Accuracy	0.990 ± 0.010	0.742 ± 0.011
Mean (average) log probability	-0.0139 ± 0.0098	-10.25 ± 0.65

Dimensionality of our feature vector is 100 (101 with bias), so any binary classification task when we have no more then 100 instances should be perfectly linearly separable. The minimisation function reached the maximum number of linear searches (8000). This is expected from logistic regression without any prior on its weights in case of linearly separable training data because the minimisation function can just draw weights to extreme values until it reaches hardware constraints (see Kevin P. Murphy. Machine Learning A Probabilistic Perspective. paragraph 8.4.3 - Gaussian approximation for logistic regression).
As a consequence, from the table it can be see that linear regression fitted the training data too well and it possibly fitted the noise as well. That is why it has extremely good accuracy and mean log probability close to zero on the training set (it is very confident in its predictions). On other hand, the result on the test set is much worse in comparison to the previous task. The mean log probability is far below the log probability of simple baseline ($log(P(y|x)) = log(0.5) = -0.6931$) which implies that in some cases our predictor gives extremely low chances (close to zero) to the right labels from the test set.

2.2 Label noise model

(a) **Modifying the likelihood**
The events $s_n = 1$ and $s_n = 0$ are mutually exclusive and together they are collectively exhaustive events and also s_n is independent from $\boldsymbol{x}$, $\boldsymbol{w}$, y that is why:

$$
\begin{aligned}
P(s_n = 1) = 1 - \epsilon \quad P(s_n = 0) = \epsilon \\
P_{uniform}(y) = \frac{1}{size(\{-1, +1\})} = \frac{1}{2} \\
P(y|\boldsymbol{x}, \boldsymbol{w}) = \sigma(y\boldsymbol{w}^T\boldsymbol{x}) \\
P(y|\boldsymbol{x}, \boldsymbol{w}, \epsilon) = P(s_n = 1)P(y|\boldsymbol{x}, \boldsymbol{w}) + P(s_n = 0)P_{uniform}(y) = \\
= (1 - \epsilon)\sigma(y\boldsymbol{w}^T\boldsymbol{x}) + \frac{\epsilon}{2}
\end{aligned}
$$

The derivatives of new model likelihood are:

$$v^{(n)} = y^{(n)}\boldsymbol{x}^{(n)T}\boldsymbol{w} \tag{1}$$

$$L(\boldsymbol{w},\epsilon) = \sum_{n=1}^{N} log(P(y^{(n)}|\boldsymbol{x^{(n)}},\boldsymbol{w},\epsilon)) = \sum_{n=1}^{N} log((1-\epsilon)\sigma(v^{(n)}) + \frac{\epsilon}{2}) \tag{2}$$

$$\nabla_{\boldsymbol{w}}\sigma(v^{(n)}) = \frac{-1}{(1+exp(-v^{(n)}))^2}exp(-v^{(n)})(-1)y^{(n)}\boldsymbol{x}^{(n)T} = \sigma^2(v^{(n)})exp(-v^{(n)})y^{(n)}\boldsymbol{x}^{(n)T} \tag{3}$$

$$\nabla_{\boldsymbol{w}}L(\boldsymbol{w},\epsilon) = \sum_{n=1}^{N} \frac{(1-\epsilon)\sigma^2(v^{(n)})exp(-v^{(n)})y^{(n)}\boldsymbol{x}^{(n)T}}{P(y^{(n)}|\boldsymbol{x^{(n)}},\boldsymbol{w},\epsilon)} \tag{4}$$

$$\frac{\partial L(\boldsymbol{w},\epsilon)}{\partial\epsilon} = \sum_{n=1}^{N} \frac{-\sigma(v^{(n)}) + \frac{1}{2}}{P(y^{(n)}|\boldsymbol{x^{(n)}},\boldsymbol{w},\epsilon)} \tag{5}$$

I have chosen the following test case for checking gradients implementation:

$$\boldsymbol{x}^{(1)} = \begin{pmatrix}0\\0\\1\end{pmatrix} \quad y^{(1)} = -1$$

$$\boldsymbol{x}^{(2)} = \begin{pmatrix}2\\2\\1\end{pmatrix} \quad y^{(2)} = 1$$

$$\boldsymbol{w} = \begin{pmatrix}1\\1\\1\end{pmatrix} \quad \epsilon = 0.2$$

I have used $h = 0.1$ which implies that absolute value of error should not exceed 0.01.
The results of checking "label noise model" likelihood derivatives via checkgrad.m are:

	derivative	finite difference	absolute error
w_1	0.0119	0.0120	0.0001
w_2	0.0119	0.0120	0.0001
w_{bias}	-0.4931	-0.4928	0.0003
ϵ	0.1818	0.1825	0.0008

As we can see in each case the absolute error doesn't exceed 0.01. As sanity check I use the fact that test case is symmetric in terms of w_1 and w_2, so our derivatives and finite differences for w_1 and w_2 must be the same.
Code snippet for "label noise model" log likelihood (will also be used in the next task):

```
function [Lp, dLp_dk] = nlm_loglike(kk, xx, yy)
    %nlm - noisy labels model
    %kk - ww + epsilon -> (D+1)x1
    %xx - NxD
    %yy - Nx1
    % Ensure labels are in {+1,-1}:
    yy  = (yy==1)*2 - 1;
    ww  = kk(1:end-1);
    eps = kk(end);

    exps = exp(-yy.*(xx*ww)); % Nx1
    sigmas = 1./(1 + exps); % Nx1
    %probabilities
    pbs = (1-eps)*sigmas + 0.5*eps; % Nx1
    Lp = sum(log(pbs));

    if nargout > 1
        %inverse probabilities
        inv_pbs = 1./pbs; % Nx1
        dLp_dw   = (inv_pbs.*(1-eps).*sigmas.^2.*exps.*yy)' * xx; % 1xD
        dLp_deps = sum(inv_pbs.*(-sigmas + 0.5)); % 1x1
        dLp_dk   = [dLp_dw, dLp_deps]'; % (D+1)x1
    end
end
```

Code snippet for this task:

```
function [] = tsk2_2_a()
    %test case
    weights = [1; 1; 1]; % x1, x2, bias % Dx1 = 3x1
    epsilon = 0.2;
    k = [weights; epsilon]; % (D+1)x1 = 4x1
    x = [0, 0, 1; 2, 2, 1]; % x1, x2, bias % NxD = 2x3
    y = [-1; 1]; % Nx1 = 2x1
    h = 0.1; % so the error should be no more 0.001

    %show not perturbed values
    [Lp, dLp_dk] = nlm_loglike(k, x, y)

    fprintf('checkgrad output:\n');
    fprintf('derivative    finite difference\n');
    accuracy = checkgrad(@nlm_loglike, k, h, x, y)
end
```

(b) **Fitting a constrained parameter**
Slightly modifying formula (5) from previous task so we can use code from previous example:

$$\frac{\partial L(\boldsymbol{w},\epsilon)}{\partial a} = \frac{\partial L(\boldsymbol{w},\epsilon)}{\partial \epsilon}\frac{\partial \epsilon}{\partial a} = \sum_{n=1}^{N} \frac{-\sigma(v^{(n)}) + \frac{1}{2}}{P(y^{(n)}|\boldsymbol{x^{(n)}},\boldsymbol{w},\epsilon)} \frac{-1}{(1+exp(-a))^2} exp(-a)(-1)$$
$$= \sum_{n=1}^{N} \frac{-\sigma(v^{(n)}) + \frac{1}{2}}{P(y^{(n)}|\boldsymbol{x^{(n)}},\boldsymbol{w},\epsilon)} \sigma^2(a) exp(-a) = \frac{\partial L(\boldsymbol{w},\epsilon)}{\partial \epsilon}\epsilon^2 exp(-a)$$

	Test set
Accuracy	