

# MLPR 2019 - Assignment 2

Vasilis Gkolemis, Sokratis Lyras

October 2019

## Question 1

We load the data and store it in appropriate numpy arrays. We don't pass the parameter `squeeze_me=True`, because we prefer our matrices to have full shape.

```
1 _filepath = os.path.abspath('../.../data/ass_02/ct_data.mat')
2 _data = io.loadmat(_filepath)
3
4 X_train = _data['X_train']
5 X_val = _data['X_val']
6 X_test = _data['X_test']
7
8 y_train = _data['y_train']
9 y_val = _data['y_val']
10 y_test = _data['y_test']
```

## Question 1a

We verify that, up to numerical rounding errors, the mean of the *y\_train* vector is zero, while this is not the case for the *y\_val*.

```
1 np.mean(y_train) # -9.13868774539957e-15
2 np.mean(y_val)   # -0.2160085093241599
```

For an array *y* we compute its mean  $\tilde{\mu}$  and standard error *s* with the following forms:

$$\tilde{\mu} = \frac{1}{N} \sum_{i=1}^N y^{(i)}$$
$$s = \sqrt{\frac{1}{N(N-1)} \sum_{i=1}^N (y^{(i)} - \tilde{\mu})^2}$$

We create a small function for implementing the above computations:

```

1 def mean_with_sterror(x):
2     m = x.mean()
3     sigma = x.std(ddof = 1)
4     sterror = sigma / np.sqrt(x.shape[0])
5     return m, sterror

```

And we apply it on  $y_{val}$  and the first 5785 elements of  $y_{train}$ :

```

1 m, err = mean_with_sterror(y_val)
2 # m = -0.2160085093241599, err = 0.01290449880016868
3 N = 5785
4 m, err = mean_with_sterror(y_train[:N])
5 # m = -0.44247687859693674, err = 0.011927303389170828

```

We summarize the results:

- $E[y_{val}] \approx -0.216 \pm 0.013$
- $E[y_{test}] \approx -0.442 \pm 0.012$  (from the first 5785 elements)

### Explanation of the misleading results

As easily observed, the estimated mean based in the first 5785 examples of the training set is misleading. This happens because the values are not randomly distributed inside the array.

We hold a simple experiment to show that if we randomly pick a subset of the training set, the results are as expected. We randomly sample  $N = 500$  (much less than 5785) examples from the training set and we compute their mean  $\mu^{(i)}$  and standard error  $s^{(i)}$ . We repeat this random process 1000 times. Finally, we compute the mean and standard deviation of the means and standard errors.

The results, for means:

$$E[\mu] = \frac{1}{1000} \sum_{i=1}^{1000} \mu^{(i)} = -0.001$$

$$s_{\mu} = \sqrt{\frac{1}{1000-1} \sum_{i=1}^{1000} (\mu_i - \mu_1)^2} = 0.045$$

and for standard error:

$$E[s] = \frac{1}{1000} \sum_{i=1}^{1000} s^{(i)} = 0.045$$

$$s_s = \sqrt{\frac{1}{1000-1} \sum_{i=1}^{1000} (s_i - \mu_s)^2} = 0.001$$

We provide the code for this small experiment:

```

1 list_m = []
2 list_stderr = []
3 np.random.seed(2)
4 N = 500
5 y_train_tmp = copy.deepcopy(y_train)
6 for i in range(1000):
7     np.random.shuffle(y_train_tmp)
8     m, err = mean_with_sterror(y_train_tmp[:N])
9     list_m.append(m)
10    list_stderr.append(err)
11
12 print("Mean estimation of y_train from %d iid samples, in 1000
13       different executions has mean: %.3f and standard deviation: %.3
14       f" %(N, np.mean(list_m), np.std(list_m, ddof=1)))
15
16 print("Standard error estimation of y_train from %d iid samples, in
17       1000 different executions has mean: %.3f and standard
18       deviation: %.3f" %(N, np.mean(list_stderr), np.std(list_stderr,
19       ddof = 1)))

```

## Question 1b

We use python, so our indices are zero-based. The code for identifying constant features:

```

1 threshold = 10e-10
2 ind_const_features = np.where(X_train.var(0) <= threshold)[0]
3 # array([ 59,  69, 179, 189, 351])

```

Constant columns are: {59, 69, 179, 189, 351}

The code for identifying duplicates:

```

1 duplicates = []
2 for j in range(X_train.shape[1]):
3     f1 = X_train[:,j]
4     tmp = X_train[:, j+1:] - np.expand_dims(f1, -1)
5     indices = np.where((np.var(tmp, 0) <= threshold))[0] + j + 1
6     duplicates.append(indices)
7
8 ind_duplicate_features = np.concatenate(duplicates).ravel()
9 ind_duplicate_features = np.sort(np.unique(ind_duplicate_features))
10 # array([ 69,  78,  79, 179, 188, 189, 199, 287, 351, 359])

```

Duplicates **later** columns are: 69, 78, 79, 179, 188, 189, 199, 287, 351, 359

## Question 2

We create a function that fits the learnable parameters to the data. Firstly, we create an augmented version of X matrix:

$$X_{aug} = \begin{bmatrix} X_{train} & \mathbf{1} \\ \sqrt{a}I_D & \mathbf{0} \end{bmatrix}$$

Afterwards we use np.linalg.lstsq routine to compute the optimal parameters  $w, b$ . This is done inside the following function:

```

1 def fit_linreg(X, yy, alpha):
2     # data augmentation
3     D = X.shape[1]
4     N = X.shape[0]
5
6     reg = np.sqrt(alpha) * np.eye(D, D)
7     X1 = np.concatenate( (X, np.ones((N, 1)) ), axis = 1)
8     reg1 = np.concatenate( (reg, np.zeros((reg.shape[1], 1)) ),
9                             axis = 1)
10    X_aug = np.concatenate( (X1, reg1), axis=0)
11    y_aug = np.concatenate( (yy, np.zeros((D, 1))), axis = 0)
12
13    # lstsq
14    W, SSE, rank, singulars = np.linalg.lstsq(X_aug, y_aug, rcond=
15                                                None)
16    W_lstsq = W[:-1]
17    b_lstsq = W[-1]
18    return W_lstsq, b_lstsq

```

We fit the model to the data using two different approaches:

- our method (fit\_linreg)
- fit\_linreg\_gradopt, which is provided on the package ct\_support\_code

```

1 # least square method
2 W_lstsq, b_lstsq = fit_linreg(X_train, y_train, 10)
3
4 # gradient method
5 alpha = 10
6 W_grad, b_grad = fit_linreg_gradopt(X_train, np.squeeze(y_train),
7                                     alpha)

```

We compute the RMSE in both cases with the following code:

```

1 def compute_RMSE(X, y, w, b):
2     # expand_dims to all single dimensional arrays
3     if len(y.shape) == 1:
4         y = np.expand_dims(y, -1)
5
6     if len(w.shape) == 1:
7         w = np.expand_dims(w, -1)
8
9     # compute RMSE
10    y_bar = np.dot(X, w) + b
11    square_erros = np.square(y_bar - y)
12    RMSE = np.sqrt(np.mean(square_erros))
13    return RMSE
14
15 RMSE_lstsq_tr = compute_RMSE(X_train, y_train, W_lstsq, b_lstsq)
16 RMSE_lstsq_val = compute_RMSE(X_val, y_val, W_lstsq, b_lstsq)
17
18 RMSE_grad_tr = compute_RMSE(X_train, y_train, W_grad, b_grad)
19 RMSE_grad_val = compute_RMSE(X_val, y_val, W_grad, b_grad)

```

We report the RMSE in both cases:

	Training	Validation
Least Squares	0.35575	0.42059
Gradient	0.35576	0.42061

We observe that the two RMSE are very close, but not identical. The gradient based optimization method cannot ensure that it will find the exact optimal, but because our cost function is convex it must converge to the optimal value. So the results are as expected.

## Question 3

### Question 3a

We use the the provided *random\_proj* routine to create the projection matrix. We create the following function for projecting the input, fitting the projected data and computing the RMSE:

```
1 def fit_and_measure_on_projection(K):
2     alpha = 10
3     proj_mat = random_proj(D, K)
4
5     # projected X
6     X_train_proj = np.dot(X_train, proj_mat)
7     X_val_proj = np.dot(X_val, proj_mat)
8
9     results = {"K": K}
10
11    # fitting
12    W_lstsq_proj, b_lstsq_proj = fit_linreg(X_train_proj, y_train,
13                                            alpha)
14
15    # RMSE
16    results['RMSE_lstsq_tr'] = compute_RMSE(X_train_proj, y_train,
17                                            W_lstsq_proj, b_lstsq_proj)
18    results['RMSE_lstsq_val'] = compute_RMSE(X_val_proj, y_val,
19                                            W_lstsq_proj, b_lstsq_proj)
20
21    return results
22
23 K = 10
24 q3a_results_proj_10 = fit_and_measure_on_projection(K)
25
26 K = 100
27 q3a_results_proj_100 = fit_and_measure_on_projection(K)
```

The errors we obtained are the following:

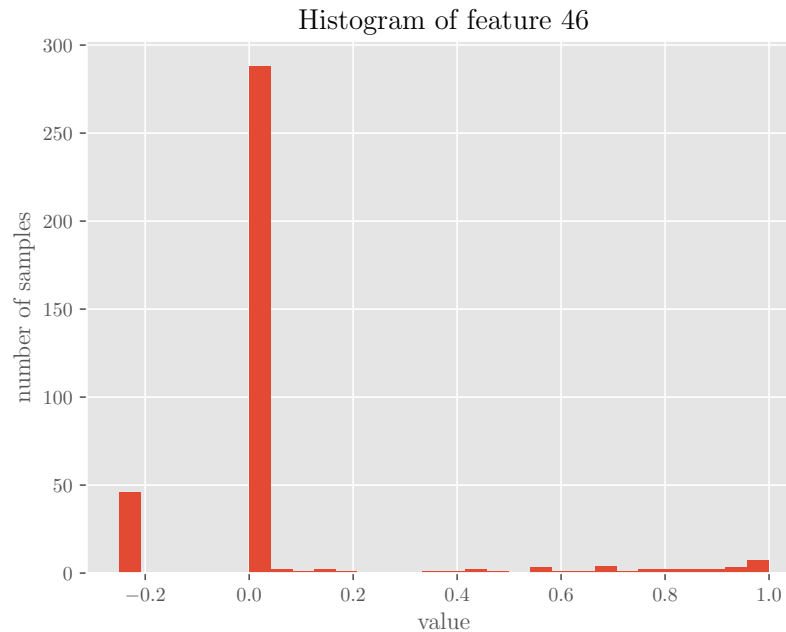
K = 10		
	Training	Validation
Least Squares	0.76299	0.80863

K = 100		
	Training	Validation
Least Squares	0.47499	0.50218

As we reduce the dimensionality, the new training error must be larger or equal than before. Since our input matrix  $X_{train}$  is full rank (as we checked), every feature gives information, so reducing the dimensionality leads to strictly higher error.

### Question 3b

It is clear that a very big fraction of feature 46 is either 0 or  $-0.25$ . More specifically, in the whole training set almost 80.36% of the examples are either 0 or  $-0.25$ .



We create a function for fitting the augmented data and measuring the RMSE:

```

1 def fit_and_measure_added_binaries():
2     alpha = 10
3
4     # projected X

```

```

5     def aug_fn(X): return np.concatenate([X, X == 0, X < 0], axis
6     =1)
7
8     X_train_aug = aug_fn(X_train)
9     X_val_aug = aug_fn(X_val)
10
11    # fitting
12    W_lstsq, b_lstsq = fit_linreg(X_train_aug, y_train, alpha)
13
14    # RMSE
15    results = {}
16    results['RMSE_lstsq_tr'] = compute_RMSE(X_train_aug, y_train,
17    W_lstsq, b_lstsq)
18    results['RMSE_lstsq_val'] = compute_RMSE(X_val_aug, y_val,
19    W_lstsq, b_lstsq)
20
21    return results
22
23 q3b_results_added_binaries = fit_and_measure_added_binaries()

```

We report the results:

	Training	Validation
RMSE	0.31783	0.37698

The training error has improved. This is expected, as the augmented  $X$  matrix:

- contains the initial  $X_{train}$  as a part of it
- has more features than  $X_{train}$
- has higher rank than  $X_{train}$  ( $= 831$ )

We also notice that the validation error has been improved as well. This is a much stronger indicator that this type of augmentation, is beneficial in terms of accuracy.

## Question 4

We fit each separate logistic regression model and store each parameters in a dictionary, with the following code:

```

1  # number of separate classification tasks
2  K = 10
3
4  # compute intervals
5  mx = np.max(y_train)
6  mn = np.min(y_train)
7  hh = (mx-mn)/(K+1)
8  thresholds = np.linspace(mn+hh, mx-hh, num=K, endpoint=True)
9
10 # set regularizer

```

```

11 alpha = 10
12
13 # init dict to store parameters
14 weight_dict = {}
15
16 # fit for each interval
17 for kk in range(K):
18     # set labels
19     labels = y_train > thresholds[kk]
20
21     # fit logistic regression
22     ww, bb = fit_logreg_gradopt(X_train, np.squeeze(labels), alpha)
23
24     # store weights
25     weight_dict[kk] = {}
26     weight_dict[kk]['w'] = ww
27     weight_dict[kk]['b'] = bb

```

We stack the learned parameters into a matrix:

```

1 weights = []
2 bias = []
3 for key, value in weight_dict.items():
4     weights.append(value["w"])
5     bias.append(value["b"])
6 ww = np.stack(weights, axis=1)
7 bb = np.expand_dims(np.stack(bias), 0)

```

We also define the sigmoid and project initial matrices to new dimension. We name the projected matrices with the suffix "\_smart":

```

1 def sigmoid(x): return 1/ (1 + np.exp(-x))
2 X_train_smart = sigmoid(np.dot(X_train, ww) + bb)
3 X_val_smart = sigmoid(np.dot(X_val, ww) + bb)
4 X_test_smart = sigmoid(np.dot(X_test, ww) + bb)

```

Finally, we fit the projected inputs to a linear regression model and compute the appropriate RMSE:

```

1 # fit linear regression
2 alpha = 10
3 W_smart, b_smart = fit_linreg(X_train_smart, y_train, alpha)
4
5 # compute errors
6 q4_RMSE_smart_tr = compute_RMSE(X_train_smart, y_train, W_smart,
7                                 b_smart)
8 q4_RMSE_smart_val = compute_RMSE(X_val_smart, y_val, W_smart,
9                                 b_smart)

```

We report the results:

	Training	Validation
RMSE	0.13825	0.25213



We observe a massive improvement in the training and validation error. Adding a non-linear projection unit before the final linear regression, has made our predictions more accurate. (Even more than the linear regression models operating in the augmented X matrix). By fitting the 10 logistic regression models, we managed to create a learned non-linear projection mechanism, that maintains more information than the random projection matrix.

## Question 5

We fit two Neural Networks with the exact same architecture and different initializations.

```

1 # random initialization
2 init_params = (np.random.randn(10), np.array(0), np.random.randn
   (10, D), np.zeros(10))
3 ww1, bb1, V1, bk1 = fit_cnn_gradopt(X_train, np.squeeze(y_train),
   10)
4 params1 = (ww1, bb1, V1, bk1)
5
6 # sophisticated initialization
7 init_params = (np.squeeze(W_smart), np.squeeze(b_smart), ww.T, np.
   squeeze(bb))
8 ww2, bb2, V2, bk2 = fit_cnn_gradopt(X_train, np.squeeze(y_train),
   10, init_params)
9 params2 = (ww2, bb2, V2, bk2)

```

We then compute the RMSE for each case:

```

1 def compute_RMSE_cnn(X, y, params):
2     y_bar = np.expand_dims(nn_cost(params, X), -1)
3     square_error = np.square(y_bar - y)
4     RMSE = np.sqrt(np.mean(square_error))
5     return RMSE
6
7 q5_RMSE_rand_tr = compute_RMSE_cnn(X_train, y_train, params1)
8 q5_RMSE_rand_val = compute_RMSE_cnn(X_val, y_val, params1)
9
10 q5_RMSE_soph_tr = compute_RMSE_cnn(X_train, y_train, params2)
11 q5_RMSE_soph_val = compute_RMSE_cnn(X_val, y_val, params2)

```

	Training	Validation
Random Init	0.09660	0.25868
Sophisticated Init	0.10435	0.25692

In the random initialization, we initialize the weights of the matrices from a normal distribution and the biases with zero. As we can see, both methods produce almost the same results. This means that the number of learnable parameters of the cnn is relatively small for the dimensionality of the problem and the amount of training data is large enough. Thus, a good local optima can easily be reached from a random initial point.

We also observe that the RMSE of the validation set of the model in question 4, is approximately the same with the RMSE of the neural network. Hence, neural network could not learn anything more than the previous model.

## Question 6

In this section, we thought of not creating a new model from scratch, but taking advantage of the work we have done so far. Thus we tried various ideas that could fit to the already implemented framework, including PCA, different weight initialization, input normalization etc. We observed that the most effective methods, in terms of the validation error, are:

- Add gaussian noise with to the training data  $X_{train} + \mathcal{N}(0, \sigma^2)$
- Change the units of the hidden layer
- Augment the input matrix  $X_{train}$  with the binary features of Question 3

The initiatives for testing these ideas was the overfitting we observed in the previous Neural Network models. That is why we added Gaussian noise to the inputs. We also thought of experimenting with adding binaries, because of the accuracy increase we observed in question 3. Finally, we tested the increasing the units of the hidden layers. This idea initially conflicts with the observed overfitting, but in combination with adding Gaussian noise, it helps our network to learn more complex representations.

We held an experiment trying all the possible combination of the hyperparameters

$$(\sigma, K, \text{adding\_binaries}) \in \{0.3, 0.4, 0.45, 0.5, 0.55, 0.6\} \times \{10, 20, 40, 60\} \times \{True, False\}$$

For each combination of the above hyperparameters, we run the training process 5 times (independently) and we computed the mean validation error of the 5 executions and the standard error. Since adding Gaussian Noise and using Random Initialization of the weights make the whole process random, this repetition helps us check the statistical significance of the different errors.

We present the results in the Figure 1:

The best model in terms of mean RMSE along the 5 different executions has hyperparameters ( $\sigma = 0.4, K = 40, \text{add\_binaries}=\text{False}$ ) and mean validation RMSE = 0.205 with standard error 0.0025.

This model has a generalisation RMSE in the test set 0.239 with standard error 0.002.

We provide the code used for holding this experiment:

```
1 def fit_and_RMSE_cnn(s, k, add_binaries, inputs):
2     def aug_fn(X): return np.concatenate([X, X == 0, X < 0], axis
      =1)
```

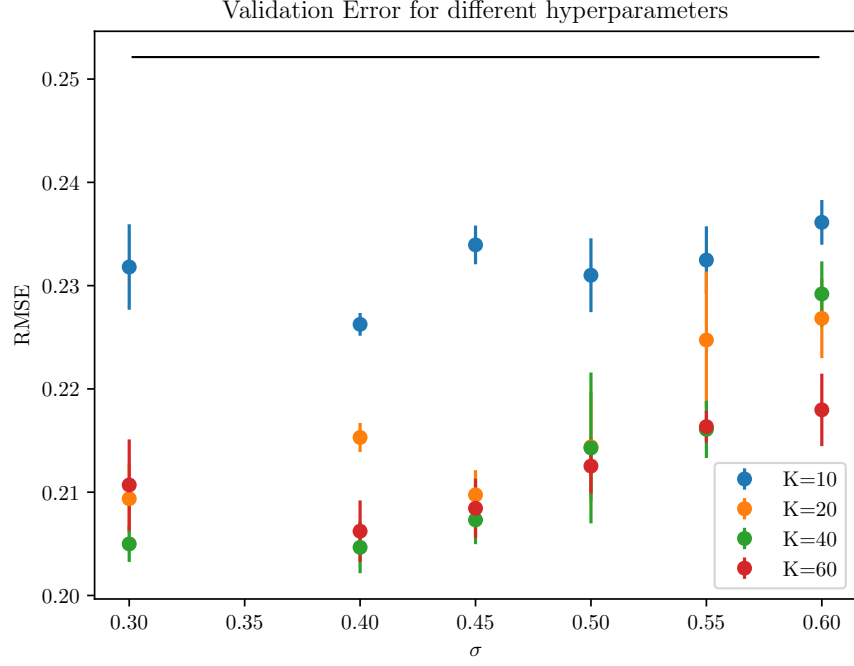


Figure 1: The black horizontal line on the top, shows the validation error of the best model of the previous questions. The error bars show the standard error of the mean RMSE across the 5 executions. We observe high overlapping so we cannot easily decide which model is the best, but we can safely suggest the the model with ( $s = 0.4, K = 40, \text{add\_binaries}=\text{False}$ ) show a significant improvement in terms of validation error from the models we had obtained so far.

```

3
4     X_train = inputs['X_train']
5     X_val = inputs['X_val']
6     X_test = inputs['X_test']
7     y_train = inputs['y_train']
8     y_val = inputs['y_val']
9     y_test = inputs['y_test']
10    D = X_train.shape[1]
11
12    alpha = 10
13    if add_binaries:
14        X_train = aug_fn(X_train)
15        X_val = aug_fn(X_val)
16        X_test = aug_fn(X_test)
17        D = 3*D
18

```

```

19     init_params = (np.random.randn(k), np.array(0), np.random.randn
20                    (k, D), np.zeros(k))
21     ww1, bb1, V1, bk1 = fit_cnn_gradopt(X_train + np.random.
22                                         standard_normal(X_train.shape) * s,
23                                         np.squeeze(y_train), alpha,
24                                         init_params)
25
26     params = (ww1, bb1, V1, bk1)
27
28     results = {}
29     results['RMSE_tr'] = compute_RMSE_cnn(X_train, y_train, params)
30     results['RMSE_val'] = compute_RMSE_cnn(X_val, y_val, params)
31     results['RMSE_test'] = compute_RMSE_cnn(X_test, y_test, params)
32     return results, params
33
34 sig = [0.3, 0.4, 0.45, 0.5, 0.55, 0.6]
35 K = [10, 20, 40, 60]
36 add_binaries = [True, False]
37 inputs = {'X_train': X_train,
38           'X_val': X_val,
39           'X_test': X_test,
40           'y_train': y_train,
41           'y_val': y_val,
42           'y_test': y_test}
43
44 results = []
45 params = []
46 hyperparams = []
47 for ii, s in enumerate(sig):
48     results.append([])
49     params.append([])
50     hyperparams.append([])
51     for jj, k in enumerate(K):
52         results[ii].append([])
53         params[ii].append([])
54         hyperparams[ii].append([])
55         for kk, bin in enumerate(add_binaries):
56             results[ii][jj].append([])
57             params[ii][jj].append([])
58             hyperparams[ii][jj].append([])
59             for n in range(5):
60                 tmp = fit_and_RMSE_cnn(s, k, bin, inputs)
61                 results[ii][jj][kk].append(tmp[0])
62                 params[ii][jj][kk].append(tmp[1])
63                 hyperparams[ii][jj][kk].append({'s':s, 'k':k, '
64 add_binaries':bin, 'n':n})

```