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 out matrices to have full shape.

```
_filepath = os.path.abspath('./../../../data/ass_02/ct_data.mat')
_data = io.loadmat(_filepath)

X_train = _data['X_train']
X_val = _data['X_val']
X_test = _data['X_test']

y_train = _data['y_train']
y_val = _data['y_val']
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*.

```
np.mean(y_train) # -9.13868774539957e-15
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:

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

And we apply it on y_val and the first 5785 elements of y_train:

```python
m, err = mean_with_sterror(y_val)
# m = -0.2160085093241599, err = 0.01290449880016868
N = 5785
m, err = mean_with_sterror(y_train[:N])
# 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:

```
list_m = []
list_stderr = []
np.random.seed(2)
N = 500
y_train_tmp = copy.deepcopy(y_train)
for i in range(1000):
    np.random.shuffle(y_train_tmp)
    m, err = mean_with_sterror(y_train_tmp[:N])
    list_m.append(m)
    list_stderr.append(err)

print("Mean estimation of y_train from %d iid samples, in 1000 different executions has mean: %.3f and standard deviation: %.3f" %(N, np.mean(list_m), np.std(list_m, ddof=1)))

print("Standard error estimation of y_train from %d iid samples, in 1000 different executions has mean: %.3f and standard deviation: %.3f" %(N, np.mean(list_stderr), np.std(list_stderr, ddof = 1)))
```

Question 1b

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

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

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

The code for identifying duplicates:

```
duplicates = []
for j in range(X_train.shape[1]):
    f1 = X_train[:,j]
    tmp = X_train[:, j+1:] - np.expand_dims(f1, -1)
    indices = np.where((np.var(tmp, 0) <= threshold))[0] + j + 1
    duplicates.append(indices)

ind_duplicate_features = np.concatenate(duplicates).ravel()
ind_duplicate_features = np.sort(np.unique(ind_duplicate_features))
# 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:

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

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

```python
def fit_linreg(X, yy, alpha):
    # data augmentation
    D = X.shape[1]
    N = X.shape[0]

    reg = np.sqrt(alpha) * np.eye(D, D)
    X1 = np.concatenate( (X, np.ones((N, 1)) ), axis = 1)
    reg1 = np.concatenate( (reg, np.zeros((reg.shape[1], 1)) ),
    axis = 1)
    X_aug = np.concatenate( (X1, reg1), axis=0)
    y_aug = np.concatenate( (yy, np.zeros((D, 1))), axis = 0)

    # lstsq
    W, SSE, rank, singulars = np.linalg.lstsq(X_aug, y_aug, rcond=
    None)
    W_lstsq = W[:-1]
    b_lstsq = W[-1]
    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

```python
# least square method
W_lstsq, b_lstsq = fit_linreg(X_train, y_train, 10)

# gradient method
alpha = 10
W_grad, b_grad = fit_linreg_gradopt(X_train, np.squeeze(y_train),
    alpha)
```

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

```python
def compute_RMSE(X, y, w, b):
    # expand_dims to all single dimensional arrays
    if len(y.shape) == 1:
        y = np.expand_dims(y, -1)

    if len(w.shape) == 1:
        w = np.expand_dims(w, -1)

    # compute RMSE
    y_bar = np.dot(X, w) + b
    square_erros = np.square(y_bar - y)
    RMSE = np.sqrt(np.mean(square_erros))
    return RMSE

RMSE_lstsq_tr = compute_RMSE(X_train, y_train, W_lstsq, b_lstsq)
RMSE_lstsq_val = compute_RMSE(X_val, y_val, W_lstsq, b_lstsq)

RMSE_grad_tr = compute_RMSE(X_train, y_train, W_grad, b_grad)
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:

```
def fit_and_measure_on_projection(K):
    alpha = 10
    proj_mat = random_proj(D, K)

    # projected X
    X_train_proj = np.dot(X_train, proj_mat)
    X_val_proj = np.dot(X_val, proj_mat)

    results = {"K": K}

    # fitting
    W_lstsq_proj, b_lstsq_proj = fit_linreg(X_train_proj, y_train,
    alpha)

    # RMSE
    results['RMSE_lstsq_tr'] = compute_RMSE(X_train_proj, y_train,
    W_lstsq_proj, b_lstsq_proj)
    results['RMSE_lstsq_val'] = compute_RMSE(X_val_proj, y_val,
    W_lstsq_proj, b_lstsq_proj)

    return results

K = 10
q3a_results_proj_10 = fit_and_measure_on_projection(K)

K = 100
q3a_results_proj_100 = fit_and_measure_on_projection(K)
```

The errors we obtained are the following:

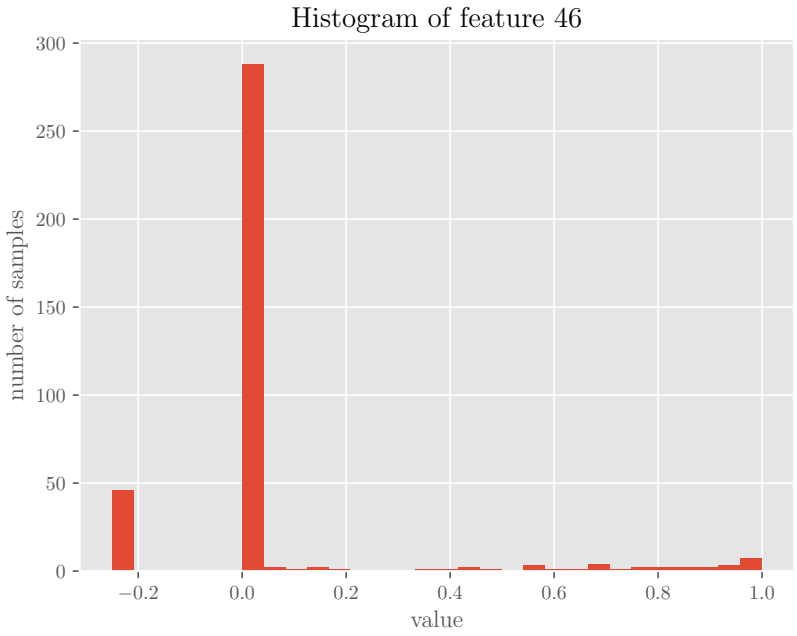
K = 10		
	Training	Validation
Least Squares	0.76299	0.80863

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:

```
def fit_and_measure_added_binaries():
    alpha = 10

    # projected X
```

```
    def aug_fn(X): return np.concatenate([X, X == 0, X < 0], axis
    =1)

    X_train_aug = aug_fn(X_train)
    X_val_aug = aug_fn(X_val)

    # fitting
    W_lstsq, b_lstsq = fit_linreg(X_train_aug, y_train, alpha)

    # RMSE
    results = {}
    results['RMSE_lstsq_tr'] = compute_RMSE(X_train_aug, y_train,
    W_lstsq, b_lstsq)
    results['RMSE_lstsq_val'] = compute_RMSE(X_val_aug, y_val,
    W_lstsq, b_lstsq)

    return results

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:

```
# number of separate classification tasks
K = 10

# compute intervals
mx = np.max(y_train)
mn = np.min(y_train)
hh = (mx-mn)/(K+1)
thresholds = np.linspace(mn+hh, mx-hh, num=K, endpoint=True)

# set regularizer
```

```
alpha = 10

# init dict to store parameters
weight_dict = {}

# fit for each interval
for kk in range(K):
    # set labels
    labels = y_train > thresholds[kk]

    # fit logistic regression
    ww, bb = fit_logreg_gradopt(X_train, np.squeeze(labels), alpha)

    # store weights
    weight_dict[kk] = {}
    weight_dict[kk]['w'] = ww
    weight_dict[kk]['b'] = bb
```

We stack the learned parameters into a matrix:

```
weights = []
bias = []
for key, value in weight_dict.items():
    weights.append(value["w"])
    bias.append(value["b"])
ww = np.stack(weights, axis=1)
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":

```
def sigmoid(x): return 1/ (1 + np.exp(-x))
X_train_smart = sigmoid(np.dot(X_train, ww) + bb)
X_val_smart = sigmoid(np.dot(X_val, ww) + bb)
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:

```
# fit linear regression
alpha = 10
W_smart, b_smart = fit_linreg(X_train_smart, y_train, alpha)

# compute errors
q4_RMSE_smart_tr = compute_RMSE(X_train_smart, y_train, W_smart,
    b_smart)
q4_RMSE_smart_val = compute_RMSE(X_val_smart, y_val, W_smart,
    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 that the random projection matrix.

Question 5

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

```
# random initialization
init_params = (np.random.randn(10), np.array(0), np.random.randn
    (10, D), np.zeros(10))
ww1, bb1, V1, bk1 = fit_cnn_gradopt(X_train, np.squeeze(y_train),
    10)
params1 = (ww1, bb1, V1, bk1)

# sophisticated initialization
init_params = (np.squeeze(W_smart), np.squeeze(b_smart), ww.T, np.
    squeeze(bb))
ww2, bb2, V2, bk2 = fit_cnn_gradopt(X_train, np.squeeze(y_train),
    10, init_params)
params2 = (ww2, bb2, V2, bk2)
```

We then compute the RMSE for each case:

```
def compute_RMSE_cnn(X, y, params):
    y_bar = np.expand_dims(nn_cost(params, X), -1)
    square_error = np.square(y_bar - y)
    RMSE = np.sqrt(np.mean(square_error))
    return RMSE

q5_RMSE_rand_tr = compute_RMSE_cnn(X_train, y_train, params1)
q5_RMSE_rand_val = compute_RMSE_cnn(X_val, y_val, params1)

q5_RMSE_soph_tr = compute_RMSE_cnn(X_train, y_train, params2)
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 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