MLPR Assignment 2 2018

s1879797

November 9, 2018

1 Question 1

1.1 Q1a

The standard error verification:

```
def compute_standard_error(y_train, y_val):
    mean_y_train = np.mean(y_train)                    # -9.13868774539957e-15
    print ("mean_y_train: ", mean_y_train)
    if round(mean_y_train, 10) == 0:
        print ("The mean is approximately 0!")
    else:
        print ("The mean is not approximately 0!")

    mean_y_val = np.mean(y_val)                        # -0.2160085093241599
    print ("mean_y_val: ", mean_y_val)
    if round(mean_y_val, 10) == 0:
        print ("The mean is approximately 0!")
    else:
        print ("The mean is not approximately 0!")

    std_y_train = np.std(y_train[:len(y_val)])
    print ("std_y_train: ", std_y_train)
    print ("std_y_train for entire y_train: ", np.std(y_train))
    std_y_val = np.std(y_val)
    print ("std_y_val: ", std_y_val)

    sem_y_train = std_y_train / np.sqrt(len(y_val))
    print ("sem_y_train: ", sem_y_train)
    sem_y_val = std_y_val / np.sqrt(len(y_val))
    print ("sem_y_val: ", sem_y_val)
```

It is true that these standard error bars sometimes could not reliably indicate the mean for future data because the actual mean of some sample is within 1 standard error 68% of time, within 2 standard errors 95% of time and within 3 standard errors 99.7% of time (**68–95–99.7 rule**). For a good estimation you have to choose good confidence intervals and I think this is the biggest problem when using standard error bars.

MEAN Y_TRAIN	STANDARD ERROR Y_TRAIN	MEAN Y_VAL	STANDARD ERROR Y_VAL
$-9.139 * 10^{-}15$	0.012	-0.216	0.013

Table 1: The mean and the standard error for training and validation

The results are presented in table 1. We can clearly see that the mean for y_train can be easily approximate to 0 and the mean for y_val is different than 0.

1.2 Q1b

The method for removing bad columns:

```
def remove_unnecessary_features(X_train, X_val, X_test):
    # Remove constant features
    X_train_delete_1 = remove_constant_features(X=X_train)
    X_val_delete_1 = remove_constant_features(X=X_val)
    X_test_delete_1 = remove_constant_features(X=X_test)

    # Remove duplicate features
    X_train_delete_2 = remove_duplicates_features(X=X_train)
    X_val_delete_2 = remove_duplicates_features(X=X_val)
    X_test_delete_2 = remove_duplicates_features(X=X_test)

    X_train_delete = X_train_delete_1 + X_train_delete_2
    X_val_delete = X_val_delete_1 + X_val_delete_2
    X_test_delete = X_test_delete_1 + X_test_delete_2

    # Get only the columns that are to be removed from all three datasets
    to_delete = reduce(np.intersect1d, (X_train_delete, X_val_delete,
    X_test_delete))

    # Filter categories
    X_train_first, X_train_second = filter_remove_categories(to_delete,
    X_train_delete_1, X_train_delete_2)
    print ("X_train first: ", X_train_first)
    print ("X_train second: ", X_train_second)
    X_val_first, X_val_second = filter_remove_categories(to_delete,
    X_val_delete_1, X_val_delete_2)
    print ("X_val first: ", X_val_first)
    print ("X_val second: ", X_val_second)
    X_test_first, X_test_second = filter_remove_categories(to_delete,
    X_test_delete_1, X_test_delete_2)
    print ("X_test first: ", X_test_first)
    print ("X_test second: ", X_test_second)

    # Delete bad columns
    X_train = np.delete(X_train, to_delete, axis=1)          # (40754, 373)
    X_val = np.delete(X_val, to_delete, axis=1)              # (5785, 373)
    X_test = np.delete(X_test, to_delete, axis=1)            # (6961, 373)
    return X_train, X_val, X_test
```

For this question I want to state that every set of data had a different number of columns that had to be removed. In order to keep the number of feature the same for every set of data I removed only the columns that had an equivalent to be removed in the other two sets. For this operation I used the function **reduce** from **functools** in order to apply **numpy intersect1d** over all three list. I had to mention this and I hope it will not be a problem since it does not influence the solution of the assignment and it does not provide a great help.

Small functions used in the above method:

```
# Remove constant input features (CATEGORY 1)
def remove_constant_features(X):
    X_delete_1 = []
    for i in range(X.shape[1]):
        if (len(set(X[:,i])) == 1):
            X_delete_1.append(i)
    return X_delete_1

# Remove duplicate input features (CATEGORY 2)
def remove_duplicates_features(X):
    unique, train_indices = np.unique(X, return_index=True, axis=1)
    X_delete_2 = list(set(range(X.shape[1])) - set(train_indices))
    return X_delete_2

# Classify removed features
def filter_remove_categories(to_delete, X_delete_1, X_delete_2):
    X_first = []
    X_second = []
    for i in to_delete:
        if i in X_delete_1:
            X_first.append(i)
        if i in X_delete_2:
            X_second.append(i)
    return X_first, X_second
```

The results are presented in table 2.

Data set	Constant Features	Duplicate Features
Training Set	[59, 69, 179, 189, 351]	[69, 78, 79, 179, 188, 189, 199, 287, 351, 359]
Validation Set	[59, 69, 179, 189, 351]	[69, 78, 79, 179, 188, 189, 199, 287, 351, 359]
Test Set	[59, 69, 179, 188, 189, 351]	[59, 69, 78, 79, 179, 188, 189, 199, 287, 351, 359]

Table 2: The columns that had to be remove from the data sets

2 Question 2

The main function for this question:

```
def set_linear_regression_baseline(X_train, X_val, y_train, y_val):
    # For training
    w_ls, b_ls = least_squares(X_train, y_train)
    err = compute_err(X=X_train, yy=y_train, ww=w_ls, bb=b_ls)
    print ("ERROR Least Squares:", err)              # 0.35524169481074713
    X_prime, y_prime, w_prime = fit_linreg(X_train, y_train, 10)
    err = compute_err(X=X_prime, yy=y_prime, ww=w_prime)
    print ("ERROR Regression with Regularization:", err)# 0.36915529643475953
    w_grad, b_grad = fit_linreg_gradopt(X_train, y_train, 10)
    err = compute_err(X=X_train, yy=y_train, ww=w_grad, bb=b_grad)
    print ("ERROR Gradient-Based Optimizer:", err)  # 0.35575973762757745

    # For validation
    err = compute_err(X=X_val, yy=y_val, ww=w_ls, bb=b_ls)
    print ("ERROR Least Squares:", err)              # 0.4182210942058271
    X_prime, y_prime, dummy = fit_linreg(X_val, y_val, 10)
    err = compute_err(X=X_prime, yy=y_prime, ww=w_prime)
    print ("ERROR Regression with Regularization:", err)# 0.47376619192358777
    err = compute_err(X=X_val, yy=y_val, ww=w_grad, bb=b_grad)
    print ("ERROR Gradient-Based Optimizer:", err)  # 0.42060375407081924
```

The **fit_linreg** function:

```
def fit_linreg(X, yy, alpha):
    D = X.shape[1]
    yy_prime = np.concatenate([yy, np.zeros(D)])
    alphaI = alpha * np.identity(D)
    X_prime = np.concatenate([X, alphaI], axis=0)
    w_prime = np.linalg.lstsq(X_prime, yy_prime, rcond=0)[0]

    return X_prime, yy_prime, w_prime
```

I also wanted to see the results for **least squares** method:

```
def least_squares(X, yy):
    X_bias = np.concatenate([np.ones((X.shape[0],1)), X], axis=1)
    w_bias = np.linalg.lstsq(X_bias, yy, rcond=0)[0];

    return w_bias[1:], w_bias[0]
```

The function which compute the predicted output:

```
def compute_err(X, yy, ww, bb=0):
    y_predicted = X.dot(ww) + bb
    return root_mean_square_error(y_expected=yy, y_predicted=y_predicted)
```

The function which computes the root mean square error:

```
def root_mean_square_error(y_expected, y_predicted):
    sum = 0
    for i in range(len(y_expected)):
        sum += ((y_expected[i] - y_predicted[i]) ** 2)

    return np.sqrt(sum/len(y_expected))
```

The result are diplayed below in table 3.

Data set	Least Squares	Linear Regression with Regularization	Gradient-Based Optimizer
Training Set	0.355	0.369	0.356
Validation Set	0.418	0.474	0.421

Table 3: Least Squares vs. Linear Regression with Regularization vs. Gradient-Based Optimizer

The results obtained from the Gradient-Based Optimizer function for root mean square error were better that the ones obtained using regularization. More than that, the results from the Gradient-Based Optimizer were similar with the ones obtained using least squares without regularization. Using a Gradient-Based Optimizer produce a lower error than Linear Regression using Regularization because for the first one the parameters are update at each iteration which can provide a faster convergence.

3 Question 3

3.1 Q3a

Overall function for this question:

```
def decrease_and_increase_input(X_train, X_val, y_train, y_val):
    # Ex 3.a

    # For training
    V = pca(X=X_train, yy=y_train, alpha=10, K=10)
    X_reduced = X_train.dot(V)
    X_prime, yy_prime, w_prime = fit_linreg(X_reduced, y_train, 10)
    err = compute_err(X=X_prime, yy=yy_prime, ww = w_prime)
    print ("Training error for K = 10: ", err)  # 0.5756376489484005
    X_reduced = X_val.dot(V)
    X_prime, yy_prime, dummy = fit_linreg(X_reduced, y_val, 10)
    err = compute_err(X=X_prime, yy=yy_prime, ww = w_prime)
    print ("Validation error for K = 10: ", err)# 0.5757941612096046

    # For validation
    V = pca(X=X_train, yy=y_train, alpha=10, K=100)
    X_reduced = X_train.dot(V)
    X_prime, yy_prime, w_prime = fit_linreg(X_reduced, y_train, 10)
    err = compute_err(X=X_prime, yy=yy_prime, ww = w_prime)
    print ("Training error for K = 100: ", err) # 0.4153790239683063
    X_reduced = X_val.dot(V)
    X_prime, yy_prime, dummy = fit_linreg(X_reduced, y_val, 10)
    err = compute_err(X=X_prime, yy=yy_prime, ww = w_prime)
    print ("Validation error for K = 100: ", err)# 0.4605743588041712

```

```
    # Ex 3.b
    X_prime, yy_prime, w_prime = histogram(X=X_train, yy=y_train, alpha=10)
    err = compute_err(X=X_prime, yy=yy_prime, ww=w_prime)
    print ("Training error: ", err)                # 0.3260341808564488
    X_prime, yy_prime, dummy = histogram(X=X_val, yy=y_val, alpha=10)
    err = compute_err(X=X_prime, yy=yy_prime, ww=w_prime)
    print ("Validation error: ", err)              # 0.38817888465014794
```

The function which uses the reduced inputs to compute the error:

```
def pca(X, yy, alpha, K):
    X_mu = np.mean(X, 0)
    X_centred = X - X_mu

    return pca_zm_proj(X=X_centred, K=K)
```

The result are diplayed below in table 4.

Data set	Linear Regression	PCA K=10	PCA K=100
Training Set	0.369	0.576	0.415
Validation Set	0.474	0.576	0.461

Table 4: Linear Regression with a normal matrix and with a reduced matrix

We use PCA in order to reduce the dimension of data so that the computation to be done faster. Normally, PCA should improve speed, accuracy and reduce error. I think that the reason that I got worse results using PCA with K = 10 than just doing Linear Regression on the normal data inputs is that PCA has some limitations and changing the number of features from 373 to 10 is too much and the results lose accuracy (for K = 100 PCA works really good and the results on validation set are better than using Linear Regression with Regularization).

3.2 Q3b

The histogram plotting function:

```
def histogram(X, yy, alpha):
    print("46th feature")
    count_0, count_25, count_0_per, count_25_per = get_percentage(X[:,46])
    print('Count: 0({0}); -0.25({1})'.format(count_0, count_25))
    print('Percentage: 0({0}); -0.25({1})'.format(count_0_per, count_25_per))
    plt.clf()
    plt.hist(X[:,45], bins= 20)
    plt.show()

    print("all data")
    count_0, count_25, count_0_per, count_25_per = get_percentage(np.ravel(X))
    print('Count: 0({0}); -0.25({1})'.format(count_0, count_25))
    print('Percentage: 0({0}); -0.25({1})'.format(count_0_per, count_25_per))
```

```
    # plt.clf()
    # plt.hist(np.ravel(X), bins= 20)
    # plt.show()

    aug_fn = lambda X: np.concatenate([X, X==0, X<0], axis=1)
    X_prime = aug_fn(X)

    print("all data after adding extra binary features"
    count_0, count_25, count_0_per, count_25_per = get_percentage(np.ravel(
    X_prime))
    print('Count: 0({0}); -0.25({1})'.format(count_0, count_25))
    print('Percentage: 0({0}); -0.25({1})'.format(count_0_per, count_25_per))
    # plt.clf()
    # plt.hist(np.ravel(X_prime), bins= 20)
    # plt.show()

    return fit_linreg(X_prime, yy, alpha)
```

The function used for counting the number of values of 1 and -0.25:

```
def get_percentage(X):
    count_0 = 0
    count_25 = 0
    for i in X:
        if i == 0:
            count_0 += 1
        elif i == -0.25:
            count_25 += 1

    count_0_per = count_0 * 100 / len(X)
    count_25_per = count_25 * 100 / len(X)
    return count_0, count_25, count_0_per, count_25_per
```

The result fractions of -0.25 and 0 regarding the training set are presented in table 5, respectively in table 6 for validation set.

Mode	-0.25 Percentage	0 Percentage
46th feature	34.38%	60.51%
All data	13.58%	66.78%
All data with extra features	4.52%	62.14%

Table 5: The fractions of -0.25 and 0 from training data

Mode	-0.25 Percentage	0 Percentage
46th feature	35.31%	56.72%
All data	14.25%	65.93%
All data with extra features	4.75%	61.92%

Table 6: The fractions of -0.25 and 0 from validation data

The new errors for training and validation sets are in table 7.

Data set	Linear Regression	Extra Binary Features
Training Set	0.369	0.326
Validation Set	0.474	0.388

Table 7: The results after adding extra binary features to input data

Adding more input features normally should decrease the error (more examples, accuracy should increase), but giving that our extra features are binary features I am not really sure of what should happen with the error. Given that we add 1 for all data that are equal to 0 and for all data lower than 0, as we can see in table 5 and table 6, the percentage of zeros from the input data is almost the same, while the percentage of -0.25 (and implicitly of all negative numbers) decreased and of course a huge number of ones has been added to the data set. Taking into account that now most of the values from the data set are 0 and 1 I think that the output has stabilized and the accuracy increased greatly which brought us a lower value for error.

The representation of the percentage for -0.25 and 0 from 46th feature is displayed in figure 2.

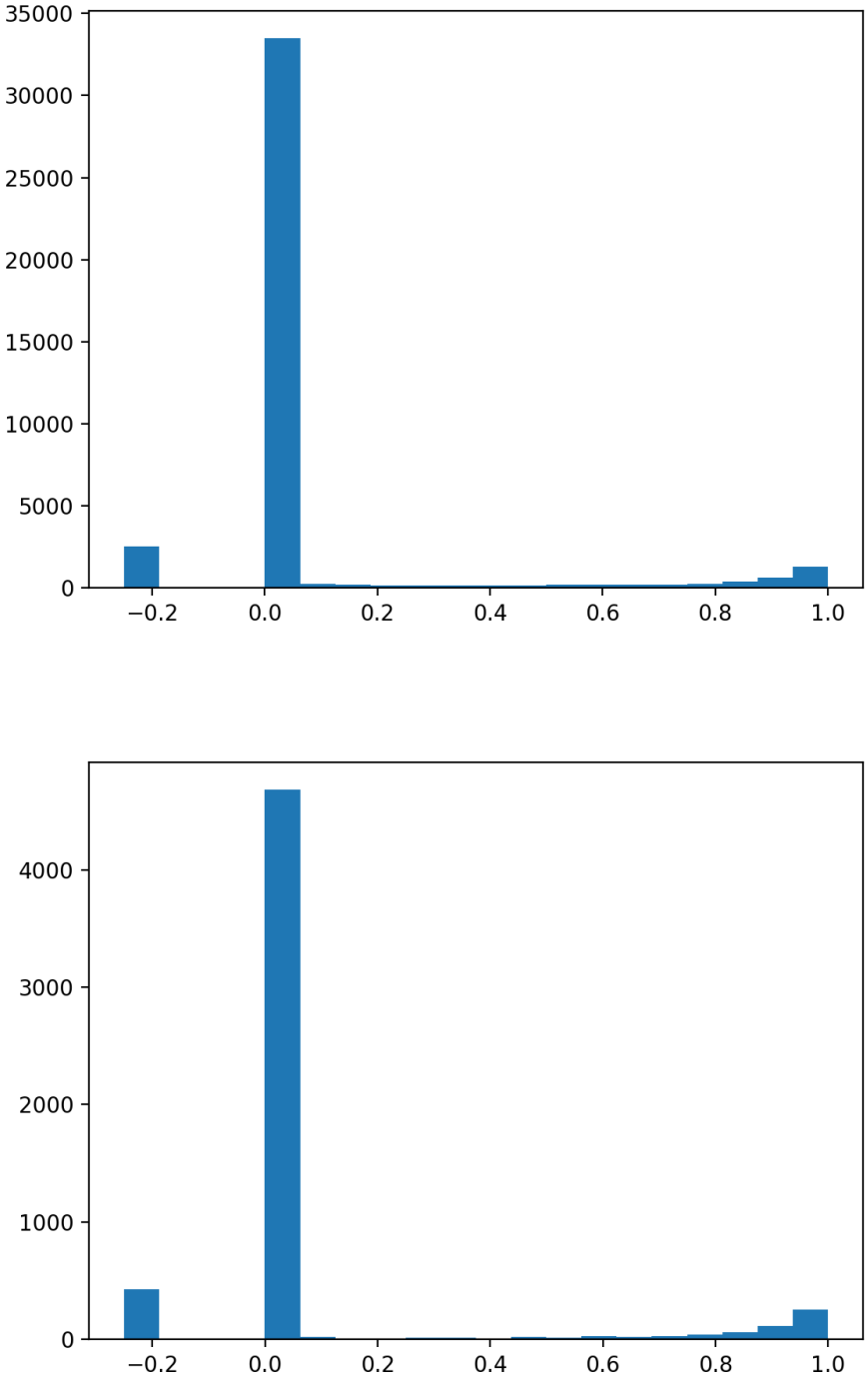

Figure 1: Histogram for training data (up) and for validation data (down)

4 Question 4

Overall function for this question:

```
def invent_classification(X_train, X_val, y_train, y_val):
    K = 10          # number of thresholded classification problems to fit
    # For training
    ww_log, bb_log = use_logreg(X=X_train, yy=y_train, alpha=10, K=10)
    X_reduced = X_train.dot(ww_log) + bb_log
    X_prime, yy_prime, w_prime = fit_linreg(X=X_reduced, yy=y_train, alpha=10)
    err = compute_err(X=X_prime, yy=yy_prime, ww = w_prime)
    print ("ERROR Logistic Regression: ", err)  # 0.42230977042363277

    # For validation
    X_reduced = X_val.dot(ww_log) + bb_log
    X_prime, yy_prime, dummy = fit_linreg(X=X_reduced, yy=y_val, alpha=10)
    err = compute_err(X=X_prime, yy=yy_prime, ww = w_prime)
    print ("ERROR Logistic Regression: ", err)  # 0.4477788975944975
```

Function added in **ct_support_code.py** in order to use the cost function for logistic regression.

```
def fit_logreg(X, yy, alpha):
    D = X.shape[1]
    args = (X, yy, alpha)
    init = (np.zeros(D), np.array(0))
    ww, bb = minimize_list(logreg_cost, init, args)

    return ww, bb
```

Apply logistic regression to reduce the input:

```
def use_logreg(X, yy, alpha, K):
    mx = np.max(yy)
    mn = np.min(yy)
    hh = (mx-mn)/(K+1)
    thresholds = np.linspace(mn+hh, mx-hh, num=K, endpoint=True)
    ww_array = []
    bb_array = []
    for kk in range(K):
        labels = yy > thresholds[kk]
        ww, bb = fit_logreg(X, labels, alpha)
        ww_array.append(ww)
        bb_array.append(bb)

    ww = np.column_stack(ww_array)
    bb = np.column_stack(bb_array)

    return ww, bb
```

I choose to compare the results for Linear Regression with Regularization, PCA with K = 10 and Logistic Regression (the input will have 10 columns so we can assume K = 10 too). As we can see in table 8, the results using Logistic Regression are better than both, PCA and Linear Regression. The PCA method will produce a new set of features based on linear combinations of the previous features (we may lose important information sometimes

and this will produce very bad results) while the Logistic Regression produce a new set of parameters based on multiple probability classes which can action on the input data changing his dimension but keeping the all the important information and providing better results, but taking more time than PCA to compute.

Data set	Linear Regression	PCA K=10	Logistic Regression
Training Set	0.369	0.576	0.422
Validation Set	0.474	0.576	0.447

Table 8: Linear Regression vs. PCA (K = 10) vs. Logistic Regression (K = 10)

5 Question 5

Overall function for this question:

```
def neural_network(X_train, X_val, y_train, y_val):
    # Random Initialization
    params = random_initialization(X_train.shape[1], 10)
    new_params = fit_nn(params=params, X=X_train, yy=y_train, alpha=10)
    y_predicted = nn_cost(params=new_params, X=X_train, alpha=10)
    err = root_mean_square_error(y_expected=y_train, y_predicted=y_predicted)
    print ("ERR Training Random:", err)            # 0.10109716170454178
    y_predicted = nn_cost(params=new_params, X=X_val, alpha=10)
    err = root_mean_square_error(y_expected=y_val, y_predicted=y_predicted)
    print ("ERR Validation Random:", err)          # 0.24988936555719002

    # Q4 Initialization
    params = q4_initialization(X_train, y_train, 10, 10)
    new_params = fit_nn(params=params, X=X_train, yy=y_train, alpha=10)
    y_predicted = nn_cost(params=new_params, X=X_train, alpha=10)
    err = root_mean_square_error(y_expected=y_train, y_predicted=y_predicted)
    print ("ERR Training Ex4:", err)                # 0.1034439502578436
    y_predicted = nn_cost(params=new_params, X=X_val, alpha=10)
    err = root_mean_square_error(y_expected=y_val, y_predicted=y_predicted)
    print ("ERR Validation Ex4:", err)              # 0.2651730161911833
```

Function which randomly initializes the parameters

```
def random_initialization(D, K):
    V = np.random.randn(K, D)
    bk = np.random.randn(K)
    ww = np.random.randn(K)
    bb = np.random.randn(1)[0]

    return (ww, bb, V, bk)
```

Function which initializes the parameters with the values from question 4.

```
def q4_initialization(X, yy, alpha, K):
    V, bk = use_logreg(X=X, yy=yy, alpha=alpha, K=K)
    X_reduced = X.dot(V) + bk
    X_prime, yy_prime, w_prime = fit_linreg(X=X_reduced, yy=yy, alpha=alpha)
    ww = w_prime
    bb = 0

    return (ww, bb, V.T, bk.T.reshape(K,))
```

Function added in **ct_support_code.py** in order to use the cost function for this neural network.

```
def fit_nn(params, X, yy=None, alpha=None):
    args = (X, yy, alpha)
    ww_bar, bb_bar, V_bar, bk_bar = minimize_list(nn_cost, params, args)

    return (ww_bar, bb_bar, V_bar, bk_bar)
```

The neural network using Question 4 parameters initialization produced very good results. Regarding using random initialization of parameters (from a normal distribution), sometimes we can get the best results, as we can see in table 9 and sometimes not, but I think it is worth the risk because even when the results are not the best, they are still comparable with using initialization from Question 4.

Data set	Linear Regression	PCA K=10	Logistic Regression	NN Random	NN Q4 Initialization
Training Set	0.369	0.576	0.422	0.101	0.103
Validation Set	0.474	0.576	0.447	0.250	0.265

Table 9: Linear Regression Regularization, PCA, Logistic Regression and Neural Network

6 Question 6

For this question I tried to implement a simple Gradient Descent algorithm because is obviously simple to implement, could converge very fast with a proper learning rate (0.05 in this case) and, as we can see in table 10, produces better results than all other methods, except for Neural Network from Question 5. Even the error value for the test set is good: 0.430. I have calculated the gradients with respect to the weights and bias using the following formulas:

$$\frac{\partial}{\partial \mathrm{m}} = \frac{2}{N}\sum_{i=1}^{N} - x_i(y_i - (mx_i + b))$$

$$\frac{\partial}{\partial \mathrm{b}} = \frac{2}{N}\sum_{i=1}^{N} - (y_i - (mx_i + b))$$

Figure 2: Partial derivatives

Data set	Linear Reg	PCA K=10	PCA K=100	Logistic Reg	NN Random	NN Q4	Gradient Descent
Training Set	0.369	0.576	0.415	0.422	0.101	0.103	0.370
Validation Set	0.474	0.576	0.461	0.447	0.250	0.265	0.432

Table 10: All models

The results using Gradient Descent are diplay in figure 3. I have also plotted the error for each iteration to make sure that the Gradient it is working correctly.

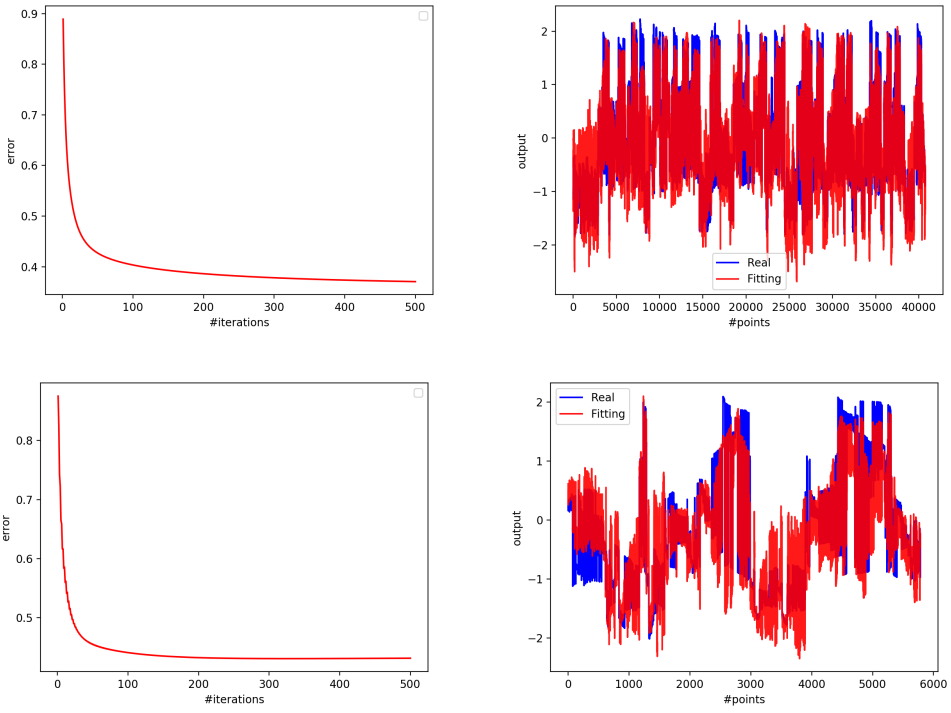

Figure 3: Error Training (top-left), Output Training (top-right), Error Validation (bottom-left), Output Validation (bottom-right)

Overall function for this question:

```
def gradient_descent(X_train, X_val, y_train, y_val, X_test, y_test):
    init = (np.zeros(X_train.shape[1]), np.array(0))
    ww, bb, params_arr = fit_gd(init, X=X_train, yy=y_train, iter=500,
    learning_rate=0.05)

    errors = plot_errors(params_arr = params_arr, X=X_train, yy=y_train)
    plot_output(X=X_train, yy=y_train, ww=ww, bb=bb)
    print ("ERROR GD Training:", errors[-1])        # 0.3704747114536849

    errors = plot_errors(params_arr = params_arr, X=X_val, yy=y_val)
    plot_output(X=X_val, yy=y_val, ww=ww, bb=bb)
    print ("ERROR GD Validation:", errors[-1])      # 0.43155709194018754

    errors = plot_errors(params_arr = params_arr, X=X_test, yy=y_test)
    plot_output(X=X_test, yy=y_test, ww=ww, bb=bb)
    print ("ERROR GD Test:", errors[-1])            # 0.4297184947308366
```

The errors plotting function:

```
def plot_errors(params_arr, X, yy):
    errors = []
    for ww, bb in params_arr:
        y_predicted = X.dot(ww) + bb
        err = root_mean_square_error(y_expected=yy, y_predicted=y_predicted)
        errors.append(err)

    grid = np.linspace(1, len(errors), len(errors))

    plt.clf()
    plt.plot(grid, errors, 'r-')
    plt.legend()
    plt.ylabel("error")
    plt.xlabel("#iterations")
    plt.show()

    return errors
```

The output plotting function:

```
def plot_output(X, yy, ww, bb=0):
    y_predicted = X.dot(ww) + bb
    grid = np.linspace(1, len(yy), len(yy))

    plt.clf()
    plt.plot(grid, yy, 'b-', label='Real')
    plt.plot(grid, y_predicted, 'r-', label='Fitting', alpha=0.9)
    plt.legend()
    plt.ylabel("output")
    plt.xlabel("#points")
    plt.show()
```

These functions were added in ct_support_code.py:
The Gradient Descent algorithm:

```
def gradient_descent(X, yy, ww, bb, learning_rate):
    bb_gradient = np.array(0.)
    ww_gradient = np.zeros(X.shape[1])
    N = X.shape[0]
    for i in range(0, N):
        x = X[i][:]
        y = yy[i]
        bb_gradient += -(2/float(N)) * (y - (x.dot(ww) + bb))
        ww_gradient += -(2/float(N)) * x * (y - (x.dot(ww) + bb))
    bb_bar = bb - (learning_rate * bb_gradient)
    ww_bar = ww - (learning_rate * ww_gradient)

    return [ww_bar, bb_bar]
```

The fitting function for Gradient Descent:

```
def fit_gd(params, X, yy=None, iter=500, learning_rate=0.01):
    ww = params[0]
    bb = params[1]
    params_arr = []
    for i in range(iter):
        print (i)
        ww, bb = gradient_descent(X, yy, ww, bb, learning_rate)
        params_arr.append((ww, bb))

    return [ww, bb, params_arr]
```