Contribution Report

Discussion and Problem Solving for Questions 1, 2, and 3:
Equal Contribution

Solutions Write-up for Questions 1,2, and 3:
David Wu

Proof-reading for Questions 1,2, and 3:
Shu Wang

Question 1.

a.

```
def logistic_predict(weights, data):
    ##############################################################################
    # TODO:                                                                      #
    ##############################################################################
    z = np.dot(data, weights[:-1]) + weights[-1]
    y = sigmoid(z)
    ##############################################################################
    #                          END OF YOUR CODE                                  #
    ##############################################################################
    return y


def evaluate(targets, y):
    ##############################################################################
    # TODO:                                                                      #
    ##############################################################################
    N, _ = targets.shape
    #To make sure it returns (float)
    ce = (-np.dot(targets[:, 0], np.log(y)[:, 0]) - np.dot(1-targets[:, 0], np.log(1-y)[:, 0]))/N

    predict = np.where(y > 0.5, 1, 0)
    frac_correct = np.count_nonzero(targets == predict)/N
    ##############################################################################
    #                          END OF YOUR CODE                                  #
    ##############################################################################
    return ce, frac_correct
```

```
def logistic(weights, data, targets, hyperparameters):

    y = logistic_predict(weights, data)
    lambd = hyperparameters["weight_regularization"]
    ##############################################################################
    # TODO:                                                                      #
    ##############################################################################
    N, _ = targets.shape
    f = evaluate(targets, y)[0] + lambd/2 * np.dot(weights[:-1].T, weights[:-1])

    df_penalty = weights.copy()
    df_penalty[-1] = 0

    data_with_dummy = np.hstack((data, np.ones((data.shape[0], 1))))
    df = np.dot(data_with_dummy.T, y-targets)/N + lambd*df_penalty
    ##############################################################################
    #                          END OF YOUR CODE                                  #
    ##############################################################################
    return f, df, y
```

b.

```
def run_logistic_regression():
    # Load all necessary datasets:
    x_train, y_train = load_train()
    # If you would like to use digits_train_small, please uncomment this line:
    #x_train, y_train = load_train_small()
    x_valid, y_valid = load_valid()
    x_test, y_test = load_test()


    hyperparameters = {
        "learning_rate": 0.1,
        "weight_regularization": 0.,
        "num_iterations": 5000
    }

    # My code
    weights = np.zeros((d + 1, 1))

    train_history = np.zeros(hyperparameters["num_iterations"])
    val_history = np.zeros(hyperparameters["num_iterations"])
    f_history = np.zeros(hyperparameters["num_iterations"])

    for t in range(hyperparameters["num_iterations"]):
        f, df, y = logistic(weights, x_train, y_train, hyperparameters)
        #For generating plots in 2c
        train_history[t] = evaluate(y_train, y)[0]
        y_val = logistic_predict(weights, x_valid)
        val_history[t] = evaluate(y_valid, y_val)[0]

        weights = weights - hyperparameters["learning_rate"] * df

    ##################### For part b ############################
    #Train Errors
    y = logistic_predict(weights, x_train)
    ce, frac_correct = evaluate(y_train, y)
    print("Training Ce:", ce)
    print("Training frac_correct:", frac_correct)

    #Compute validation error
    y = logistic_predict(weights, x_valid)
    ce, frac_correct = evaluate(y_valid, y)

    print("Validation Ce:", ce)
    print("Validation frac_correct:", frac_correct)
```

```
#Compute Test Error after selecting the models
y = logistic_predict(weights, x_test)
ce, frac_correct = evaluate(y_test, y)
print("Test Ce:", ce)
print("Test frac_correct:", frac_correct)
```

Num_Iterations = 1000

Learning Rate	Model Output
0.03	Training Ce: 0.06862187416226331 Training frac_correct: 0.985 Validation Ce: 0.08024806933672739 Validation frac_correct: 0.985
0.05	Training Ce: 0.053698059031191556 Training frac_correct: 0.99 Validation Ce: 0.06886940815529113 Validation frac_correct: 0.98
0.1	Training Ce: 0.03729025534051304 Training frac_correct: 0.9933333333333333 Validation Ce: 0.05921871110578798 Validation frac_correct: 0.98

Num_Iterations = 5000

Learning Rate	Model Output
0.03	Training Ce: 0.02934998866393638 Training frac_correct: 0.9966666666666667 Validation Ce: 0.05582774755313737 Validation frac_correct: 0.98
0.05	Training Ce: 0.02092527956116547 Training frac_correct: 0.9983333333333333 Validation Ce: 0.05329672558411321 Validation frac_correct: 0.98
0.1	Training Ce: 0.0123229413952326 Training frac_correct: 1.0 Validation Ce: 0.05238292389414841 Validation frac_correct: 0.98

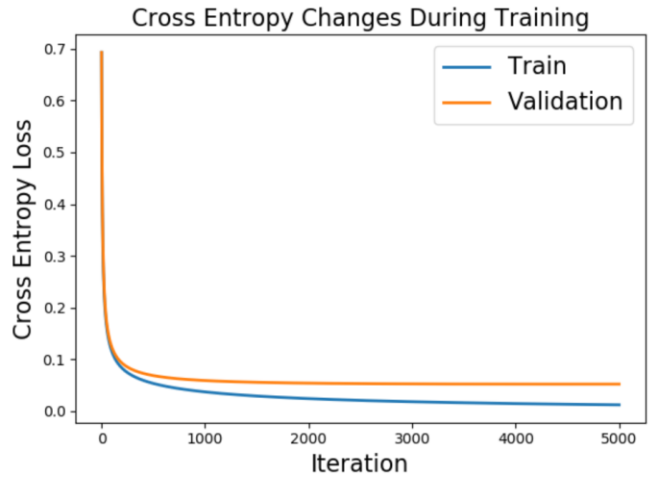
After trying some more combinations (not shown here), it seems that a learning rate of 0.1 and 5000 iterations is a good choice of parameters. In terms of the validation set, it has the lowest cross entropy loss and had predicts 98% of the cases correctly.

```
Test Ce: 0.06373342325818517
Test frac_correct: 0.97
```

c.

```
# For plots in part c
plt.plot(train_history, linewidth=2, label='Train')
plt.plot(val_history, linewidth=2, label='Validation')
plt.legend(loc='upper right', fontsize = 16)
plt.title("Cross Entropy Changes During Training", fontsize = 16)
plt.ylabel("Cross Entropy Loss", fontsize = 16)
plt.xlabel("Iteration", fontsize = 16)
plt.tight_layout()
plt.show()
```

The plot below shows that the CE loss has mostly converge after 5000 iterations. Therefore, adding more iterations will not be helpful.

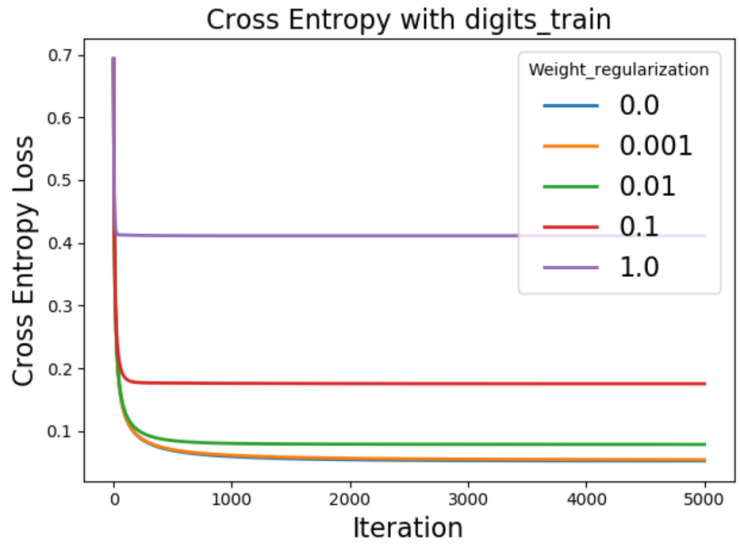


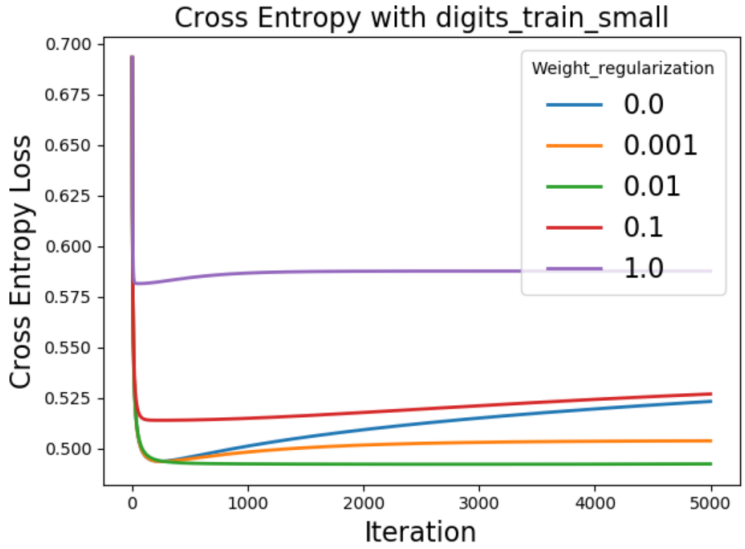
Running the code multiple times did not change the results. This is because we did not introduce randomness in the optimization algorithm. If we initialized the weights randomly or used stochastic gradient ascent, we may see different results across iterations.

d.

```
################### For part d ###################################
val_history = np.zeros((5, hyperparameters["num_iterations"]))
lambd_array = np.array([0, 0.001, 0.01, 0.1, 1.0])
for i in range(len(lambd_array)):
    hyperparameters["weight_regularization"] = lambd_array[i]
    weights = np.zeros((d + 1, 1))
    for t in range(hyperparameters["num_iterations"]):
        f, df, y = logistic(weights, x_train, y_train, hyperparameters)

        y_val = logistic_predict(weights, x_valid)
        val_history[i, t] = evaluate(y_valid, y_val)[0]

        weights = weights - hyperparameters["learning_rate"] * df
```

```
plt.plot(val_history[0], linewidth=2, label= lambd_array[0])
plt.plot(val_history[1], linewidth=2, label= lambd_array[1])
plt.plot(val_history[2], linewidth=2, label= lambd_array[2])
plt.plot(val_history[3], linewidth=2, label= lambd_array[3])
plt.plot(val_history[4], linewidth=2, label= lambd_array[4])

plt.legend(title = "Weight_regularization", loc='upper right', fontsize=16)
plt.title("Cross Entropy with digits_train_small", fontsize=16)
plt.ylabel("Cross Entropy Loss", fontsize=16)
plt.xlabel("Iteration", fontsize=16)
plt.tight_layout()
plt.show()
```





e. We see that for the larger training data set (data_train), cross entropy loss is increasing as lambda increases. Since the training data set has a lot of data points, we are less likely to over fit

the data. Adding a larger penalty term reduces the value of some of the weights and increases the bias of our model. As a result, the model does not generalize well.

Test accuracy and cross entropy for $\lambda = 0$ is same as part b.

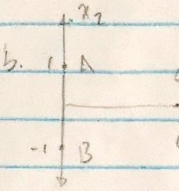
For the small data set, see that validation loss first decreases and then increases λ increases. The optimal weight regularization is 0.1. Since the dataset it small, it is easy to over fit. As a result, adding some penalty term is helpful to avoid overfitting. However, when λ is too large, the model becomes biased and cannot fit the details of the training set well.

Using the $\lambda = 0.1$ model, we get the test accuracy is:

```
Test Ce: 0.5568298442811063
Test frac_correct: 0.705
```

2. Label the points as follows: $A = x^{(1)} = (0, 1)$, $B = x^{(2)} = (0, -1)$ and $C = x^{(3)} = (4, 0)$.

a. The optimal clustering (A, B, C). All points must belong to some cluster. Since there is only 1 cluster, all points must be in the same cluster.

b. 

Optimal Cluster $\boxed{(A, B, C)}$

Initialize $m_1 = (\frac{4}{3}, 0)$ and $m_2 = (1000, 1000)$. Clearly $r^{(i)} = (1, 0)$ for all i. Refitting m_1 (m_2 is empty cluster), we get $m_1 = \frac{1}{3}\sum_{i=1}^{3} x^{(i)} = (\frac{1}{3}[0+0+4], \frac{1}{3}(1+(-1)+0)) = (\frac{4}{3}, 0)$.

Therefore this is an optimal configuration and (A, B, C) is an optimal clustering.

Optimal Cluster: $\boxed{(A, B), (C)}$ Initialize $m_1 = (0, 0)$ and $m_2 = (4, 0)$

$d(A, m_1)^2 = d(B, m_1)^2 = 0 + 1^2 = 1$ $d(A, m_2)^2 = d(B, m_2)^2 = 4^2 + 1^2 = 17$,

so A, B are assigned to m_1.

$d(C, m_1)^2 = 4^2 + 0^2 = 16$, $d(C, m_2)^2 = 0$, so C is assigned to m_2.

Refitting: $m_1 = \frac{1}{2}(0+0, 1+(-1)) = (0, 0)$. $m_2 = (4, 0)$.

Therefore this is a optimal configuration.

Finally, (A, C) (B) cannot be an optimal clustering. If it is, then the optimal configuration has AC assigned to m_1 and B assigned to m_2. Under Euclidean distance, this implies $m_1 = \frac{1}{2}(0+4, 1+0) = (2, 0.5)$ and $m_2 = B = (0, -1)$. However:

$d(A, m_1)^2 = 2^2 + 0.5^2 = 4.25$ and $d(A, m_2)^2 = 0^2 + 2^2 = 4$.

Therefore the above is not an optimal configuration, and we have a contradiction. By symmetry (B, C), (A) cannot be an optimal clustering.

c. Consider refitting a center $m = (m_1, m_2)$ with n points assigned to it. We want $\min_m \sum_{i=1}^{n} d(x^{(i)}, m)^2 = \min_m \sum_{i=1}^{n} \left(\frac{x_1 - m_1}{4}\right)^2 + (x_2 - m_2)^2$.

Taking partials with respect to m_1 and m_2 and setting equal to 0 gives (and simplifying):

2.a $\sum 2\left(\frac{x_1 - m_1}{4}\right)\cdot\frac{1}{4} = 0$ and $\sum x_2 - m_2 = 0$, so

$m_1 = \bar{x}_1$ and $m_2 = \bar{x}_2$ are the critical points, where $\bar{}$ represents mean.

Since quadratics are convex, $m = (\bar{x}_1, \bar{x}_2)$ yields a minimum.

Optimal Clusters: $\boxed{(A,B,C)}$ Similar to b, initialize $m_1 = x^{(1)} = (\frac{4}{3}, 0)$, and $m_2 = (1000, 1000)$, gives an optimal configuration.

$\boxed{(A,B),(C)}$ Initialize $m_1 = (0,0)$, $m_2 = (4,0)$.

$d(A,m_1)^2 = d(B,m_1)^2 = 1$ $\quad d(A,m_2)^2 = d(B,m_2)^2 = (\frac{4}{4})^2 + 1^2 = 2$, so A, B are assigned to m_1. Refitting same as part b $\Rightarrow$ optimal configuration.

$\boxed{(A,C),(B)}$ Initialize $m_1 = (2, 0.5)$ and $m_2 = (0, -1)$.

$d(A,m_1) = (\frac{2}{4})^2 + 0.5^2 = 0.5$ $\quad d(A,m_2) = 2^2 = 4$

$d(C,m_1) = (\frac{2}{4})^2 + 0.5^2 = 0.5$ and $d(C,m_2) = (\frac{4}{4})^2 + 1^2 = 2$, so

A, C are assigned to m_1 and B is assigned to m_2.

m_1 is same after refitting since $\frac{1}{2}(4+0, 1+0) = (2, 0.5)$. Similarly, refitting m_2 gives $m_2 = B = (0, -1)$. $\Rightarrow$ optimal configuration

$\boxed{(B,C),(A)}$ By symmetry, this is also an optimal clustering.

Question 3.

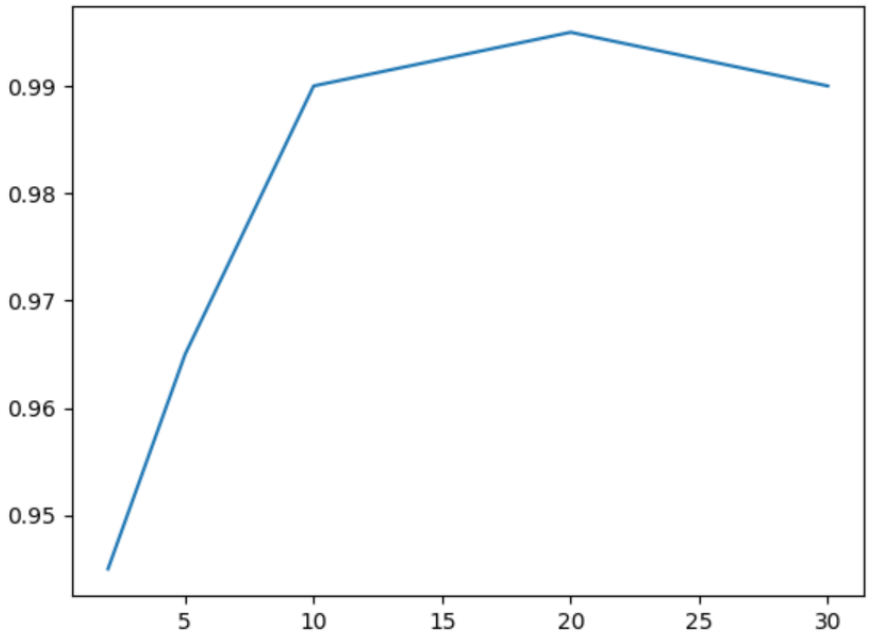
a.

```
def pca(x, k):
    n, d = x.shape
    ###########################################################################
    # TODO:                                                                   #
    ###########################################################################
    one_over_N = np.full((n, 1), 1/n)
    mean = np.dot(x.T, one_over_N)

    #Calculating covariance
    ones = np.ones((n,1))
    temp = x - np.dot(ones, mean.T)
    covariance = 1/n * np.dot(temp.T, temp)

    #Eigenvalues and Eigenvectors
    value, v = lin.eigh(covariance, subset_by_index = [d-k, d-1])

    proj_x = np.dot(v.T, temp.T)
    ###########################################################################
    #                           END OF YOUR CODE                              #
    ###########################################################################
    return v, mean, proj_x
```

b.

```
def pca_classify():
    # Load all necessary datasets:
    x_train, y_train = load_train()
    x_valid, y_valid = load_valid()
    x_test, y_test = load_test()

    # Make sure the PCA algorithm is correctly implemented.
    v, mean, proj_x = pca(x_train, 5)
    # The below code visualize the eigenvectors.
    show_eigenvectors(v)

    ###########################################################################
    # TODO:                                                                   #
    ###########################################################################
    k_lst = [2, 5, 10, 20, 30]
    val_acc = np.zeros(len(k_lst))
    for j, k in enumerate(k_lst):
        v, mean, proj_x_train = pca(x_train, k)

        temp = x_valid - mean.T
        proj_x_val = np.dot(v.T, temp.T)
```

```
for i in range(x_valid.shape[0]):
    # For each validation sample, perform 1-NN classifier on
    # the training code vector.

    #Make matrix such that each column is proj_train_i - proj_val_i
    difference = proj_x_train - proj_x_val[:, i].reshape((-1,1))

    #Calculating the euclidean distance
    difference = difference ** 2

    #Currently the ith element of a column stores (proj_train_i_i - proj_val_i)^2
    #Add the columns together to get euclidean distance squared
    distance = np.dot(difference.T, np.ones((k,1)))

    #distance is a N_train x 1 matrix
    closest_point = np.argmin(distance)

    if y_train[closest_point] == y_valid[i]:
        val_acc[j] += 1/y_valid.shape[0]
```



c. We pick the model with the highest validation accuracy. Looking at the plot between K and val_accuracy, we pick K = 20.

Note: If we wanted faster runtimes or to save memory, we could pick a smaller K (for example K = 10). However, the code currently runs quite fast regardless of K, so this isn't a big concern.

d. Fitting the model on the test data with K = 20, we get:

```
Test accuracy: 0.9899999999999899
```

This is slightly higher than the accuracy from logistic regression which was around 0.97