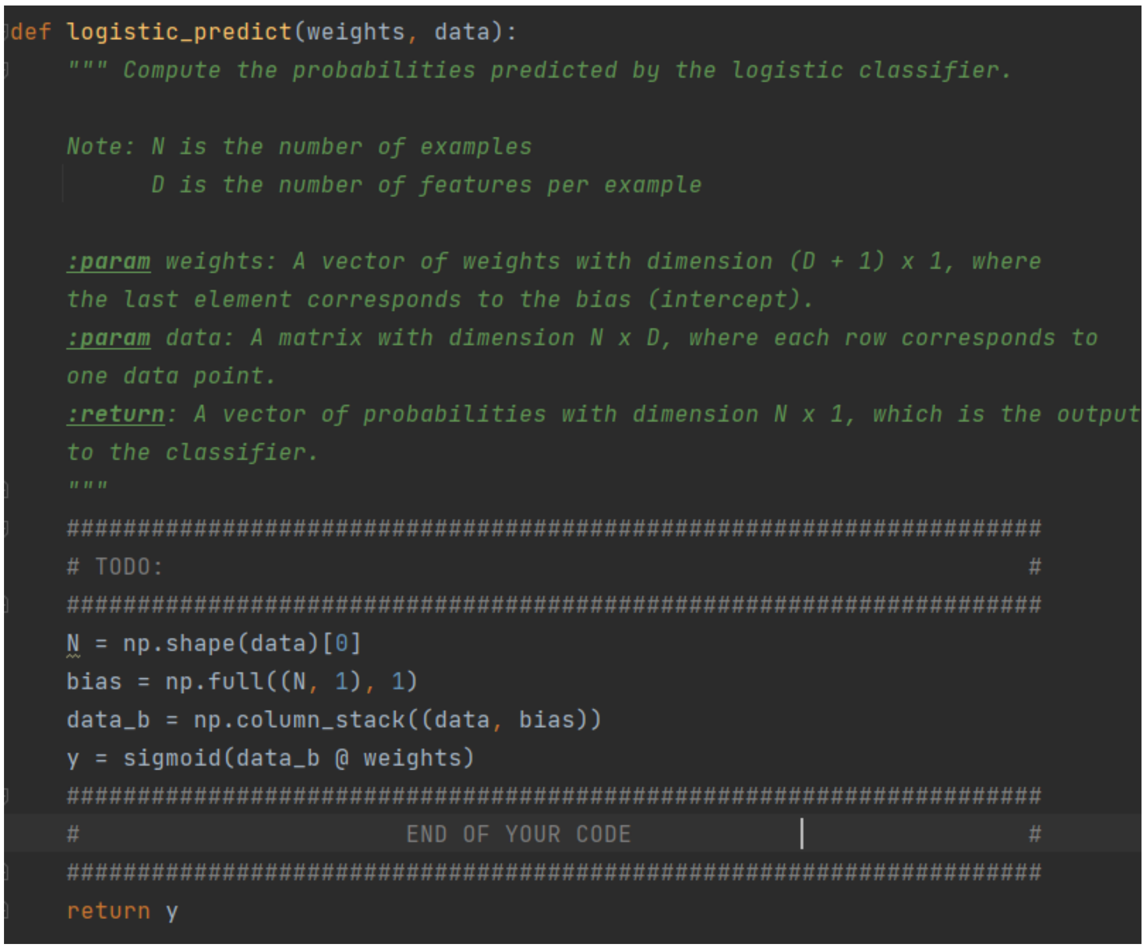
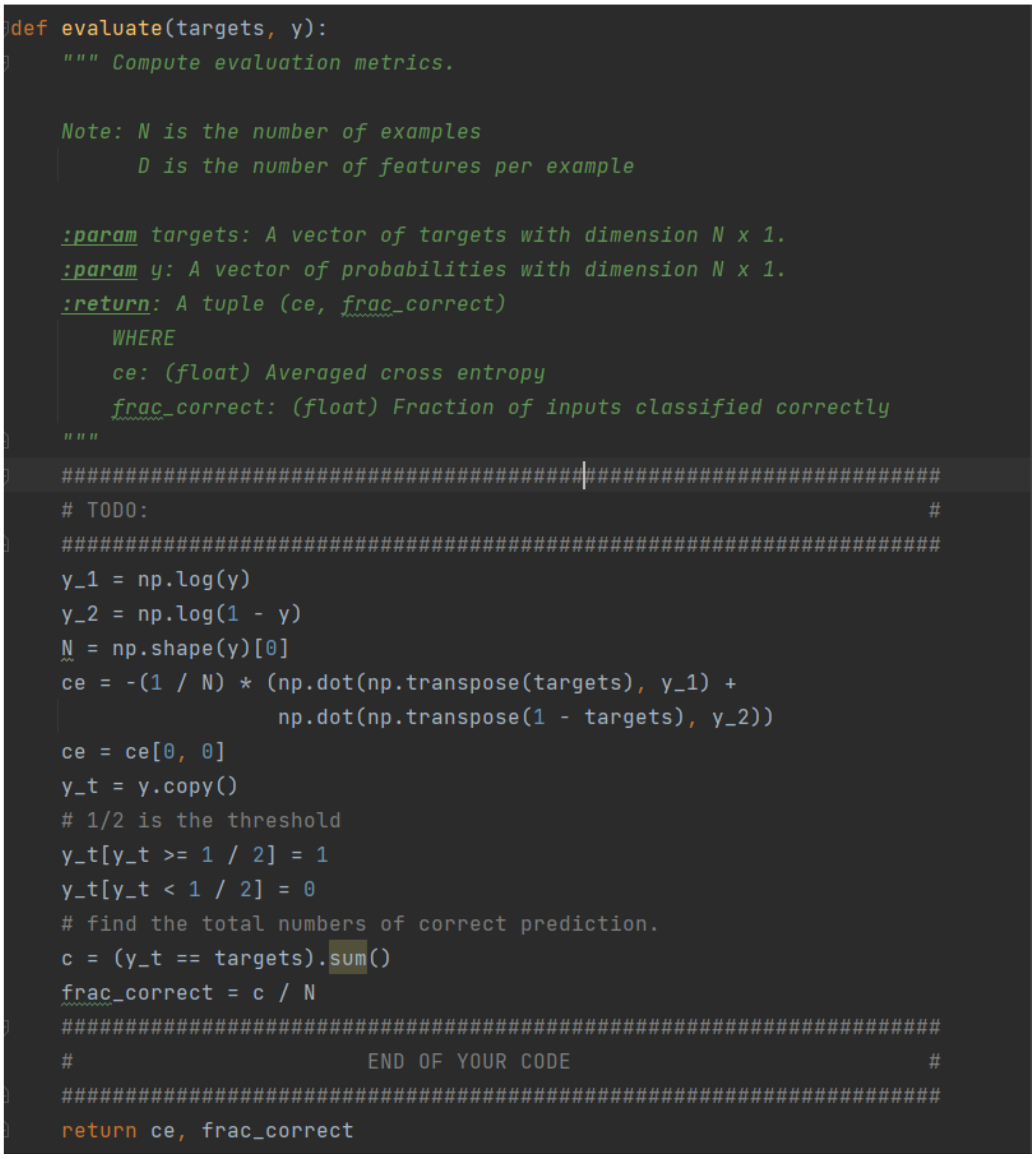
1 a)



```python
def logistic_predict(weights, data):
    """ Compute the probabilities predicted by the logistic classifier.

    Note: N is the number of examples
          D is the number of features per example

    :param weights: A vector of weights with dimension (D + 1) x 1, where
    the last element corresponds to the bias (intercept).
    :param data: A matrix with dimension N x D, where each row corresponds to
    one data point.
    :return: A vector of probabilities with dimension N x 1, which is the output
    to the classifier.
    """
    #####################################################################
    # TODO:                                                             #
    #####################################################################
    N = np.shape(data)[0]
    bias = np.full((N, 1), 1)
    data_b = np.column_stack((data, bias))
    y = sigmoid(data_b @ weights)
    #####################################################################
    #                       END OF YOUR CODE                            #
    #####################################################################
    return y
```

Function evaluate is on the next page.

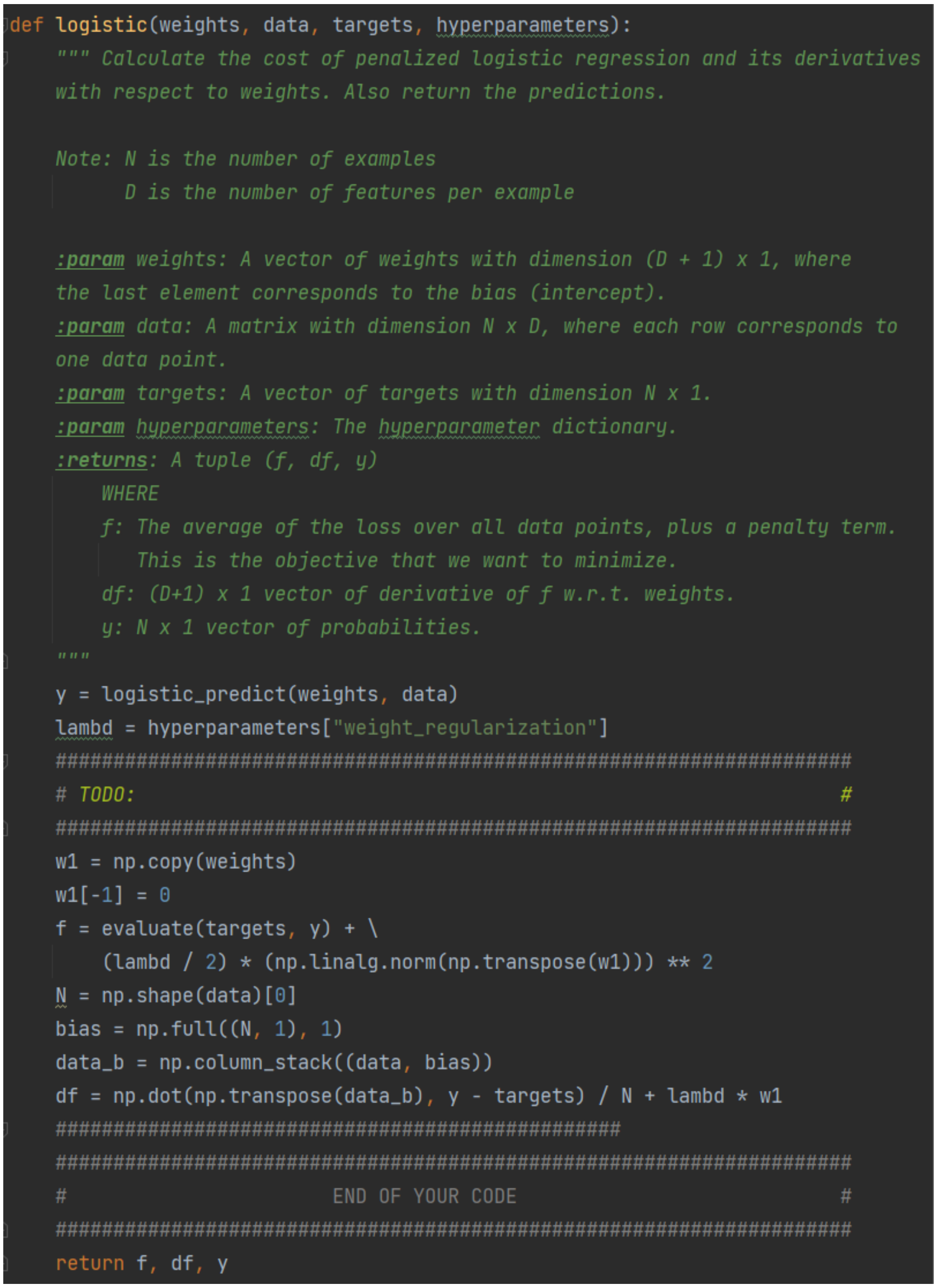
1a)



```
def evaluate(targets, y):
    """ Compute evaluation metrics.

    Note: N is the number of examples
          D is the number of features per example

    :param targets: A vector of targets with dimension N x 1.
    :param y: A vector of probabilities with dimension N x 1.
    :return: A tuple (ce, frac_correct)
        WHERE
        ce: (float) Averaged cross entropy
        frac_correct: (float) Fraction of inputs classified correctly
    """
    #####################################################################
    # TODO:                                                             #
    #####################################################################
    y_1 = np.log(y)
    y_2 = np.log(1 - y)
    N = np.shape(y)[0]
    ce = -(1 / N) * (np.dot(np.transpose(targets), y_1) +
                     np.dot(np.transpose(1 - targets), y_2))
    ce = ce[0, 0]
    y_t = y.copy()
    # 1/2 is the threshold
    y_t[y_t >= 1 / 2] = 1
    y_t[y_t < 1 / 2] = 0
    # find the total numbers of correct prediction.
    c = (y_t == targets).sum()
    frac_correct = c / N
    #####################################################################
    #                        END OF YOUR CODE                           #
    #####################################################################
    return ce, frac_correct
```

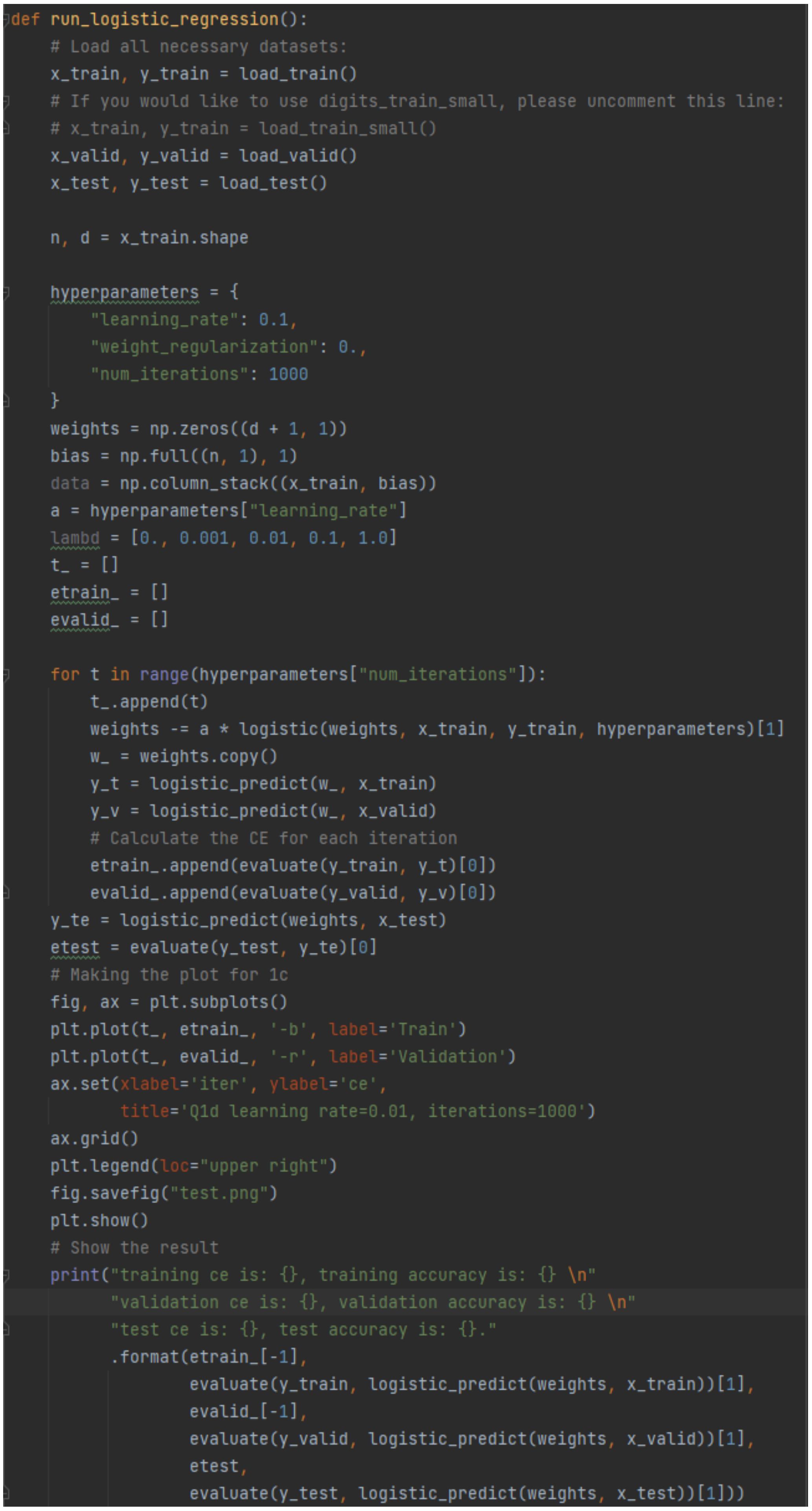
Function logistic is on the next page.

1a)



```python
def logistic(weights, data, targets, hyperparameters):
    """ Calculate the cost of penalized logistic regression and its derivatives
    with respect to weights. Also return the predictions.

    Note: N is the number of examples
          D is the number of features per example

    :param weights: A vector of weights with dimension (D + 1) x 1, where
    the last element corresponds to the bias (intercept).
    :param data: A matrix with dimension N x D, where each row corresponds to
    one data point.
    :param targets: A vector of targets with dimension N x 1.
    :param hyperparameters: The hyperparameter dictionary.
    :returns: A tuple (f, df, y)
        WHERE
        f: The average of the loss over all data points, plus a penalty term.
           This is the objective that we want to minimize.
        df: (D+1) x 1 vector of derivative of f w.r.t. weights.
        y: N x 1 vector of probabilities.
    """
    y = logistic_predict(weights, data)
    lambd = hyperparameters["weight_regularization"]
    #####################################################################
    # TODO:                                                             #
    #####################################################################
    w1 = np.copy(weights)
    w1[-1] = 0
    f = evaluate(targets, y) + \
        (lambd / 2) * (np.linalg.norm(np.transpose(w1))) ** 2
    N = np.shape(data)[0]
    bias = np.full((N, 1), 1)
    data_b = np.column_stack((data, bias))
    df = np.dot(np.transpose(data_b), y - targets) / N + lambd * w1
    ##################################################
    #####################################################################
    #                       END OF YOUR CODE                            #
    #####################################################################
    return f, df, y
```

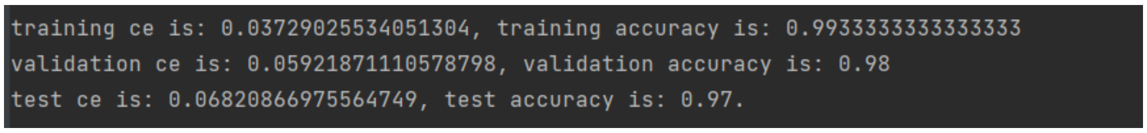
1b)



```python
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()

    n, d = x_train.shape

    hyperparameters = {
        "learning_rate": 0.1,
        "weight_regularization": 0.,
        "num_iterations": 1000
    }
    weights = np.zeros((d + 1, 1))
    bias = np.full((n, 1), 1)
    data = np.column_stack((x_train, bias))
    a = hyperparameters["learning_rate"]
    lambd = [0., 0.001, 0.01, 0.1, 1.0]
    t_ = []
    etrain_ = []
    evalid_ = []

    for t in range(hyperparameters["num_iterations"]):
        t_.append(t)
        weights -= a * logistic(weights, x_train, y_train, hyperparameters)[1]
        w_ = weights.copy()
        y_t = logistic_predict(w_, x_train)
        y_v = logistic_predict(w_, x_valid)
        # Calculate the CE for each iteration
        etrain_.append(evaluate(y_train, y_t)[0])
        evalid_.append(evaluate(y_valid, y_v)[0])
    y_te = logistic_predict(weights, x_test)
    etest = evaluate(y_test, y_te)[0]
    # Making the plot for 1c
    fig, ax = plt.subplots()
    plt.plot(t_, etrain_, '-b', label='Train')
    plt.plot(t_, evalid_, '-r', label='Validation')
    ax.set(xlabel='iter', ylabel='ce',
           title='Q1d learning rate=0.01, iterations=1000')
    ax.grid()
    plt.legend(loc="upper right")
    fig.savefig("test.png")
    plt.show()
    # Show the result
    print("training ce is: {}, training accuracy is: {} \n"
          "validation ce is: {}, validation accuracy is: {} \n"
          "test ce is: {}, test accuracy is: {}."
          .format(etrain_[-1],
                  evaluate(y_train, logistic_predict(weights, x_train))[1],
                  evalid_[-1],
                  evaluate(y_valid, logistic_predict(weights, x_valid))[1],
                  etest,
                  evaluate(y_test, logistic_predict(weights, x_test))[1]))
```

More contents for 1b) on the next page

1b) I chose the learning rate to be 0.1 and the number of iterations to be 1000. The cross entropy and accuracy are shown below.



```
training ce is: 0.03729025534051304, training accuracy is: 0.9933333333333333
validation ce is: 0.05921871110578798, validation accuracy is: 0.98
test ce is: 0.06820866975564749, test accuracy is: 0.97.
```

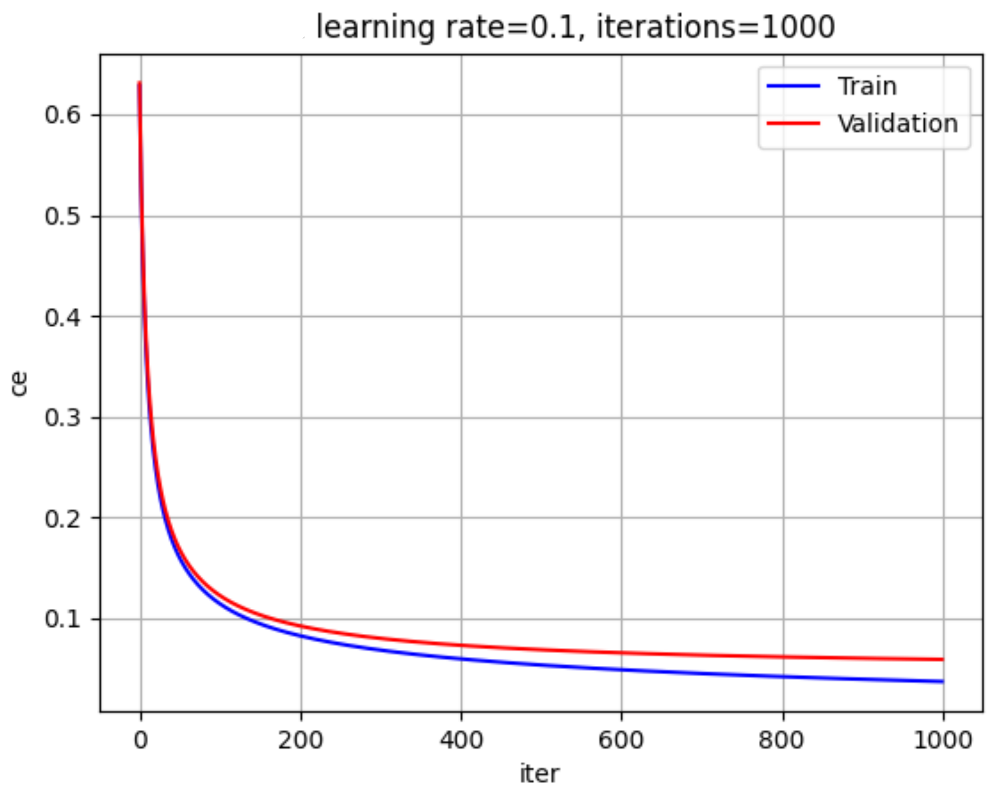
I have tried several hyperparameters settings. Below are some of the examples I tried.

I have tried to set the learning rate to be 0.01 and 1 and keep the same number of iterations. When the learning rate is 1, I found that the training cross entropy decreases, and the training accuracy increases. While for the validation set, the cross entropy decreases only by 0.01, and the accuracy decreases. So, the improvement when I change the learning rate to 1 is not too significant, while it is likely to be too large so that the test error can be oscillating and not be able to converge. So, I prefer 0.1 to 1. When I set the learning rate to be 0.01, the validation cross entropy doubles, and the validation accuracy deceases. So, this learning rate is too small for the error to converge. So, I chose the learning rate to be 0.1.

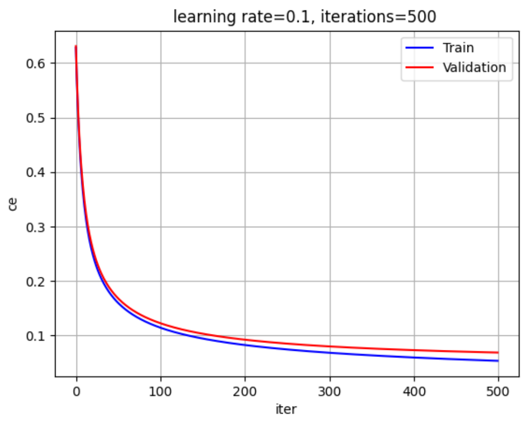
In terms of the number of iterations. I have tried to set the number of iterations to be 500, 1000, and 2000 and keep the learning rate to be 0.1. When the number of iterations in 500, validation cross entropy decreases, and accuracy remains the same. So, I chose 1000 over 500. When the number of iterations is 2000, the validation cross entropy and accuracy did not change a lot. While iterations of 2000 make it takes more time to compute, so I chose 1000 over 2000.

As a result, after numbers of trials, I chose the learning rate to be 0.1 and the number of iterations to be 1000.

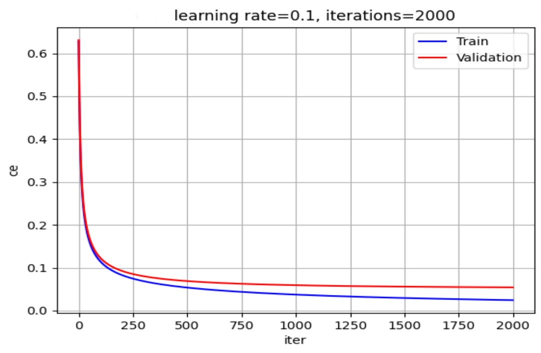
1(c)



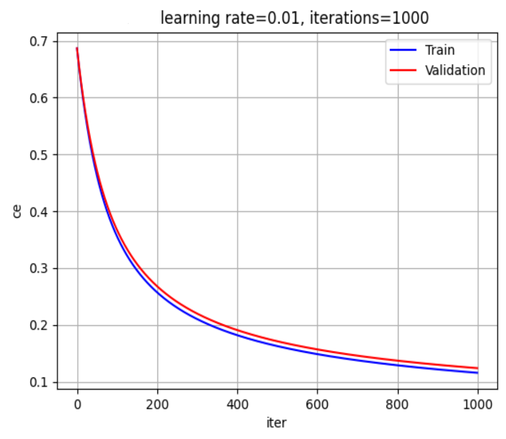
The plot above shows the cross entropy vs. number of iteration curves when the learning rate is 0.1, the number of iterations is 1000, and the L2 regularization is 0, which is the hyperparameter setting I use for the model. There are several plots below shows the plot with different hyperparameter setting that I did not choose at the end.



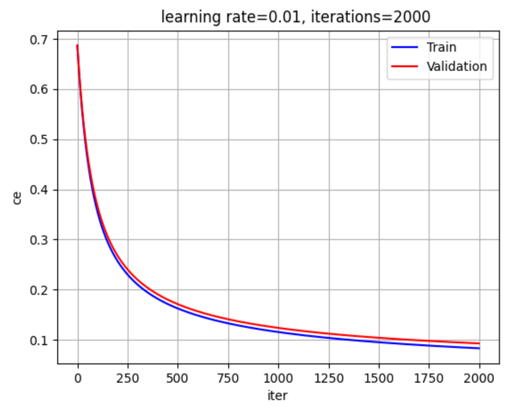
The plot on the left is when learning rate is 0.1 and number of iterations is 500. We can see that the cross entropy of validation set when the iterations is 500 is less than when the iterations is 1000. So I choose number of iterations to be 1000.



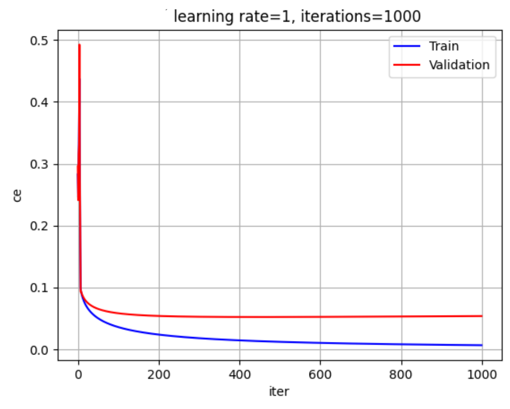
The plot on the left is when learning rate is 0.1 and number of iterations is 2000. We can see that the cross entropy of the validation set did not decrease a lot. So, I choose number of iterations to be 1000.



The plot on the left is when the learning rate is 0.01 and the number of iterations is 1000. The cross entropy of the training and validation sets is greater than when the learning rate is 0.1. Besides, the cross entropy is not converging.



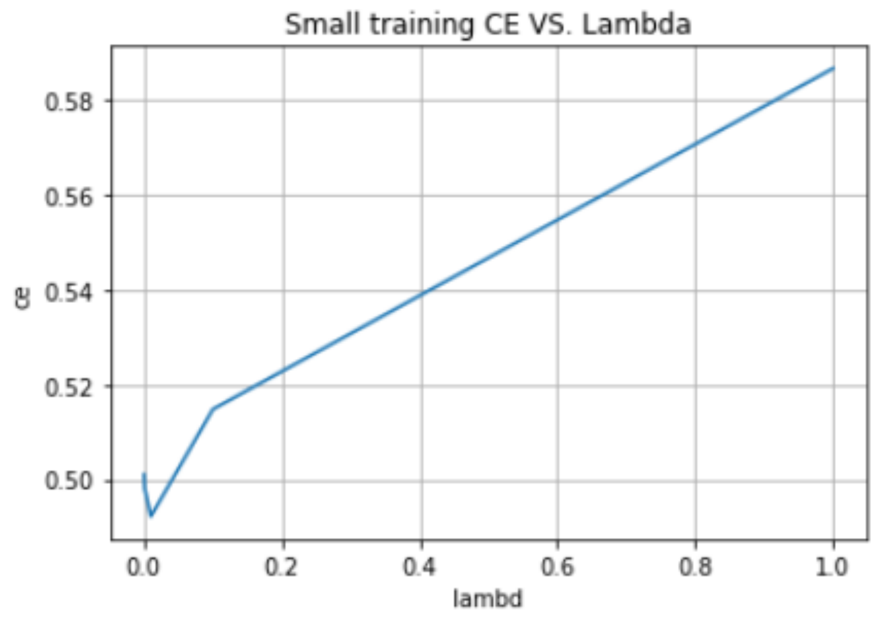
The plot still has the same issue when I tuned the number of iterations up to 2000 and keep the learning rate as 0.01

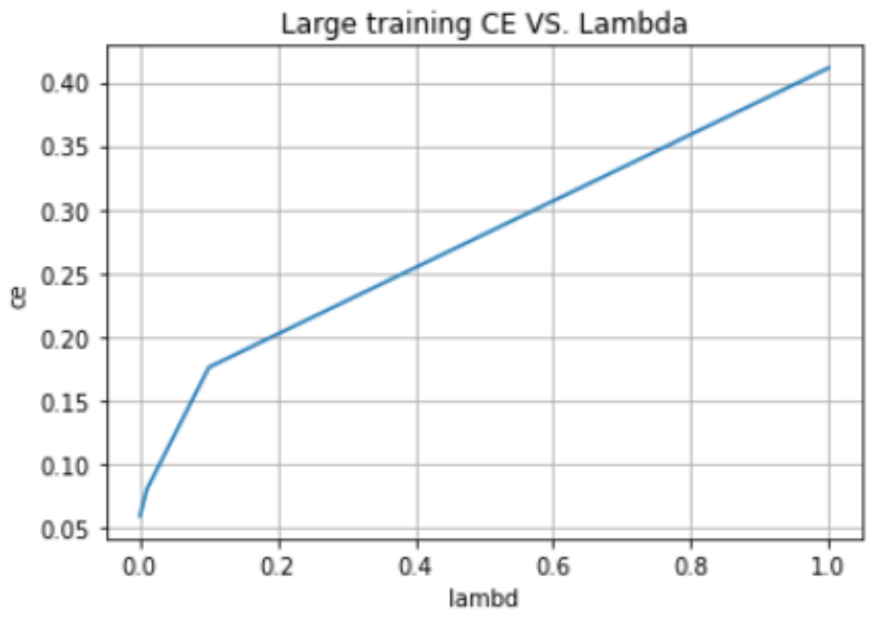


When the learning rate is 1 and number of iterations is 1000. The cross entropy oscillates a lot at first and there is a huge difference between the cross entropy of the training set and validation set at the end.

As a result, I chose the learning rate to be 0.1 to avoid oscillation as well as to make sure the convergence of the cross entropy. Besides, I chose the number of iterations to be 1000 so that the cross entropy will gradually convergent while saving the time of calculations.

1d)





Hyperparameter setting for both Training sets are the same: learning rate = 0.1
iterations = 1000

1 e) For the small training set, we can see that the cross entropy decreases first then go up. The cross entropy of the large training set goes up. Then after a certain lambda, the increasing rate slows down.

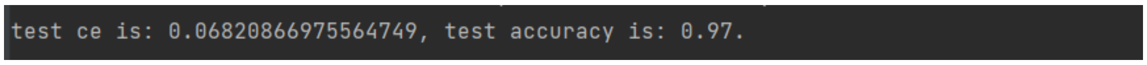
Lambda is used to increase the loss if the weights is too large, and large weights lead to overfit and the model will be less generalized.

For the small training set, the reason of the cross entropy decreases at first is due to lack of training data will cause the model to be less generalized, hence the weights will be large, and the model will be overfit. Besides, lambda will make the model less overfit and make the model more generalized. As a result, the cross entropy decreases at first. While the cross entropy increases after a certain lambda, since when lambda is too large, the model will be underfit which reduces the cross entropy.

For the large training set, the cross entropy increases at the beginning since the training set is large, so the model can be generalized. The weights are not too large, and the model is not overfit. While increasing lambda will make the weights too small hence underfit the model and increases the cross entropy to the validation set.

As a result, I chose lambda to be zero, since I will train the model with the larger training set, and for the large training set, zero lambda gives the best cross entropy.

The test cross entropy and accuracy are shown below:



```
test ce is: 0.06820866975564749, test accuracy is: 0.97.
```

2 let $X^{(1)} = (0, 1)$, $X^{(2)} = (0, -1)$, $X^{(3)} = (4, 0)$

m_i denoted as the ith cluster center

2a) $\{(0,1), (0,-1), (4,0)\}$ in one cluster with center $(4/3, 0)$

First, pick m_1 at $(4/3, 0)$

Then since $k=1$, so $r_1^{(1)}, r_1^{(2)}, r_1^{(3)} = 1$

so $m_1 = \frac{(0,1)+(0,-1)+(4,0)}{3} = (4/3, 0)$

2b) There are two possible partitions

- $\{(0,1), (0,-1)\}$ in one cluster with center $(0,0)$
  $\{(4,0)\}$ in the other cluster with center $(4,0)$

- $\{(0,1), (0,-1), (4,0)\}$ in one cluster with center $(4/3, 0)$
  There is an empty cluster

① For $\{(0,1), (0,-1)\}$ in one cluster with center $(0,0)$

$\{(4,0)\}$ in the other cluster with center $(4,0)$

- Pick $m_1 = (0,0)$, $m_2 = (2,0)$ at first
- Then $\hat{k}^{(1)} = \arg\min_k \|m_k - X^{(1)}\|^2 = 1$

  (since $\|m_1 - (0,1)\|^2 = 1 < \|m_2 - (0,1)\|^2 = 5$)

  Similarly $\hat{k}^{(2)} = 1$

  $\hat{k}^{(3)} = 2$

  so $r_1^{(1)} = 1$ $r_1^{(2)} = 1$ $r_2^{(3)} = 1$

- so $m_1 = \frac{(0,1)+(0,-1)}{2} = (0,0)$

  $m_2 = \frac{(4,0)}{1} = (4,0) = X_3 \Rightarrow$ converge

② For $\{(0,1), (0,-1), (4,0)\}$ in one cluster with center $(4/3, 0)$

There is an empty cluster

- Pick $m_1 = (0,0)$, $m_2 = (10,10)$ at first
- Then $r_1^{(1)} = r_1^{(2)} = r_1^{(3)} = 1$, $r_0^{(1)} = r_0^{(2)} = r_0^{(3)} = 0$
- so $m_1 = \frac{(0,1)+(0,-1)+(4,0)}{3} = (4/3, 0)$

And there is an empty cluster

2c) There are four possible partitions:

- $\{(0,1),(0,-1)\}$ in one cluster with center $(0,0)$

  $\{(4,0)\}$ in the other cluster with center $(4,0)$
- $\{(0,1),(0,-1),(4,0)\}$ in one cluster with center $(4/3,0)$

  There is an empty cluster
- $\{(4,0),(0,1)\}$ in one cluster with center $(2,1/2)$

  $\{(0,-1)\}$ in the other with center $(0,-1)$
- $\{(4,0),(0,-1)\}$ in one cluster with center $(2,-1/2)$

  $\{(0,1)\}$ in the other with center $(0,1)$

① For $\{(0,1),(0,-1)\}$ in one cluster with center $(0,0)$

$\{(4,0)\}$ in other with center $(4,0)$

- Pick $m_1=(0,0)$, $m_2=(4,0)$ at first
- Then $\hat{k}^{(1)} = \underset{k}{\arg\min}\left(\sqrt{\left(\frac{0-y_1}{4}\right)^2+(1-y_2)^2}\right)^2$

  $d((0,1),(0,0))^2 = \left(\sqrt{\left(\frac{0-0}{4}\right)^2+(1-0)^2}\right)^2 = 1$

  $d((0,1),(4,0))^2 = \left(\sqrt{\left(\frac{4-0}{4}\right)^2+(1-0)^2}\right)^2 = 2>1$

  so $\hat{k}^{(1)}=1$, $r_1^{(1)}=1$

  $d((0,-1),(0,0))^2 = \left(\sqrt{\left(\frac{0-0}{4}\right)^2+(-1-0)^2}\right)^2 = 1$

  $d((0,-1),(4,0))^2 = \left(\sqrt{\left(\frac{4-0}{4}\right)^2+(-1-0)^2}\right)^2 = 2>1$

  so $\hat{k}^{(2)}=1$, $r_1^{(2)}=1$

  $d((4,0),(4,0))^2 = \left(\sqrt{\left(\frac{4-4}{4}\right)^2+(0-0)^2}\right)^2 = 0$, so $r_2^{(3)}=1$
- To find $\min \sum_{n=1}^{3}\sum_{k=1}^{2} r_k^{(n)}\left(\left(\frac{x_1^{(n)}-m_1^{(k)}}{4}\right)^2+(x_2^{(n)}-m_2^{(k)})^2\right)$

  Set $\partial/\partial m_1^{(j)} \sum_{n=1}^{3}\sum_{k=1}^{2} r_k^{(n)}\left(\left(\frac{x_1^{(n)}-m_1^{(k)}}{4}\right)^2+(x_2^{(n)}-m_2^{(k)})^2\right) = \sum_{n=1}^{3} r_j^{(n)}\frac{x_1^{(n)}-m_1^{(j)}}{2}\cdot(-1)=0$

  so $m_1^{(j)} = \sum_{n=1}^{3}\frac{r_j^{(n)}x_1^{(n)}}{2} \Big/ \sum_{n=1}^{3} r_j^{(n)}/2$

  Set $\partial/\partial m_2^{(j)} \sum_{n=1}^{3}\sum_{k=1}^{2} r_k^{(n)}\left(\left(\frac{x_1^{(n)}-m_1^{(k)}}{4}\right)^2+(x_2^{(n)}-m_2^{(k)})^2\right) = \sum_{n=1}^{3}(-2r_j^{(n)}(x_2^{(n)}-m_2^{(j)}))=0$

  so $\sum_{n=1}^{3} r_j^{(n)}x_2^{(n)} - \sum_{n=1}^{3} r_j^{(n)}m_2^{(j)} = 0$

  so $m_2^{(j)} = \frac{\sum_{n=1}^{3} r_j^{(n)}x_2^{(n)}}{\sum_{n=1}^{3} r_j^{(n)}}$
- so $m_j = \frac{\sum_n r_j^{(n)}x^{(n)}}{\sum_n r_j^{(n)}}$

  so $m_1 = \frac{\binom{0}{1}+\binom{0}{-1}}{2} = (0,0)$ $\quad m_2 = \frac{\binom{4}{0}}{1} = (4,0)$

② For $\{(0,1), (0,-1), (4,0)\}$ in one cluster with center $(4/3, 0)$

There is an empty cluster

- Pick $m_1 = (4/3, 0)$, $m_2 = (0, 10)$ at first
- $d((4/3,0),(0,1))^2 = d((4/3,0),(0,-1))^2 = 10/9$

  $d((0,10),(0,1))^2 = 81$

  $d((0,10),(0,-1))^2 = 121$

  so $r_1^{(1)} = r_1^{(2)} = 1$

  $d((4,0),(4/3,0))^2 = (4 - 4/3)^2 = 4/9$

  $d((4,0),(0,10))^2 = 1 + 100 = 101$

  so $r_1^{(3)} = 1$

  so $m_1 = (4,0)+(0,1)+(0,-1)/3 = (4/3, 0)$

③ For $\{(4,0), (0,1)\}$ in one cluster with center $(2, 1/2)$

$\{(0,-1)\}$ in the other with center $(0,-1)$

- Pick $m_1 = (2, 1/2)$, $m_2 = (0,-1)$
- $d((4,0),(2,1/2))^2 = (4-2/4)^2 + (0-1/2)^2 = 1/2$

  $d((4,0),(0,-1))^2 = (4-0/4)^2 + 1^2 = 2$

  so $r_1^{(3)} = 1$

  $d((0,1),(2,1/2))^2 = (2/4)^2 + (1/2)^2 = 1/2$

  $d((0,1),(0,-1))^2 = 4$

  so $r_1^{(1)} = 1$

  $d((0,-1),(0,-1)) = 0$

  so $r_2^{(2)} = 1$

- $m_1 = (4,0)+(0,1)/2 = (2, 1/2)$

  $m_2 = (0,-1)/1 = (0,-1)$

④ For $\{(4,0), (0,-1)\}$ in one cluster with center $(2,-1/2)$

$\{(0,1)\}$ in the other with center $(0,1)$

- Pick $m_1 = (2,-1/2)$, $m_2 = (0,1)$
- $d((4,0),(2,-1/2))^2 = (4-2/4)^2 + (0+1/2)^2 = 1/2$

  $d((4,0),(0,1))^2 = (4-0/4)^2 + 1^2 = 2$

  so $r_1^{(3)} = 1$

  $d((0,1),(0,1))^2 = 0$

  so $r_2^{(1)} = 1$

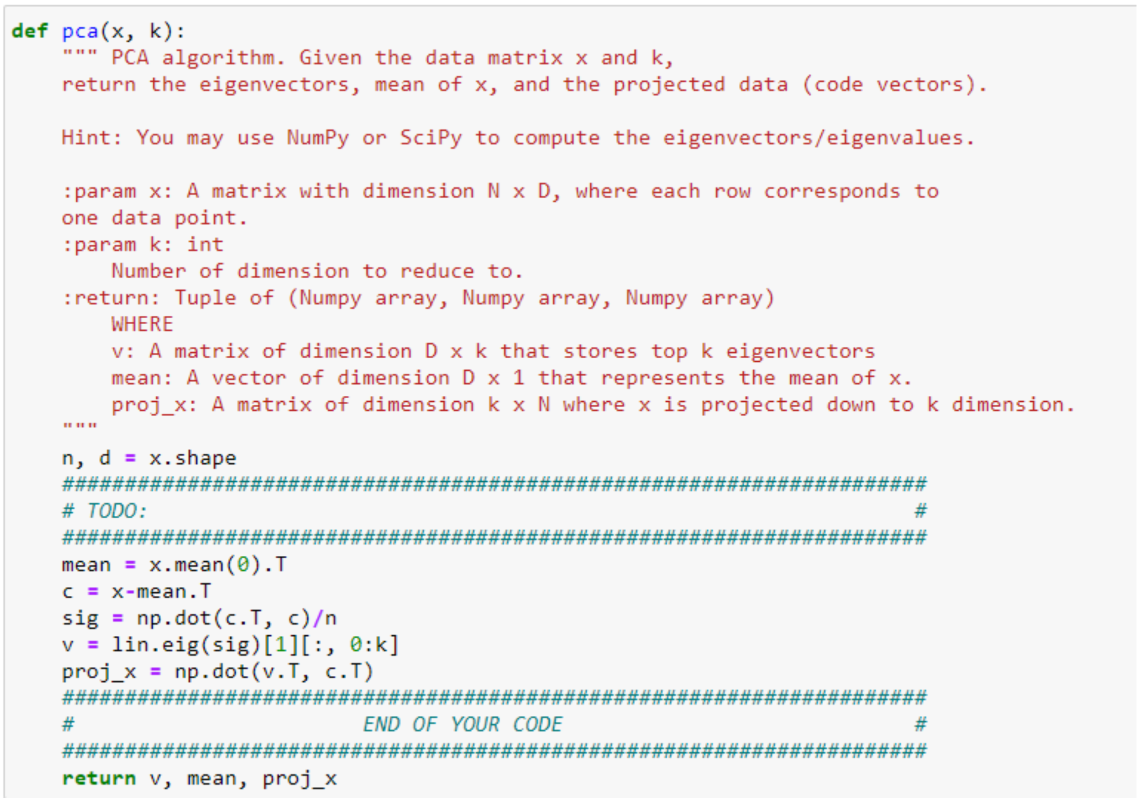
  $d((0,-1),(2,-1/2)) = (2/4)^2 + (-1+1/2)^2 = 1/2$

  $d((0,-1),(0,1)) = 4$

  so $r_1^{(2)} = 1$
- $m_1 = (4,0)+(0,-1)/2 = (2,-1/2)$

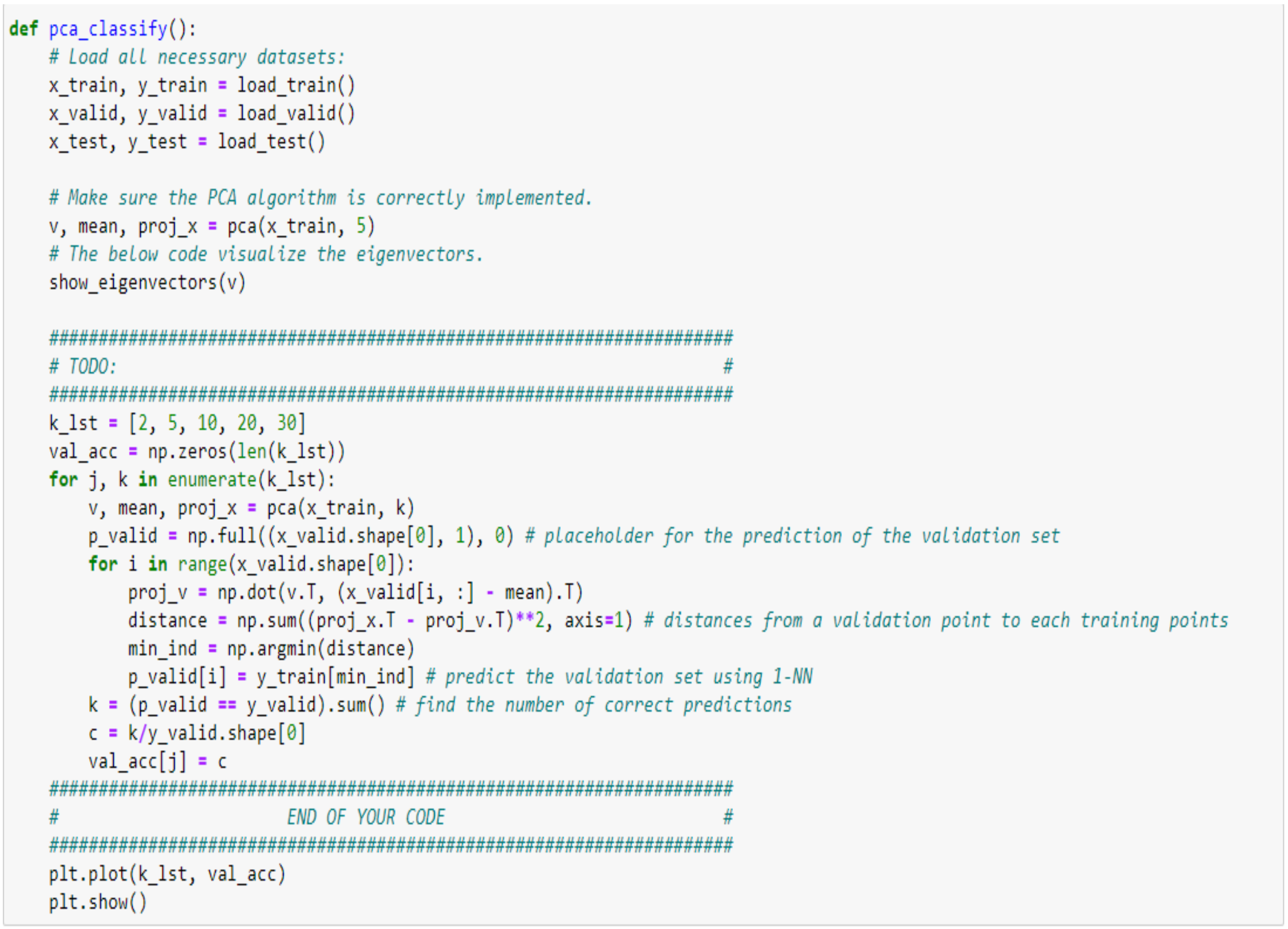
  $m_2 = (0,1)/1 = (0,1)$

3 a)

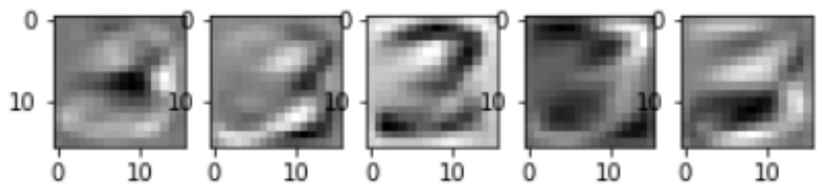


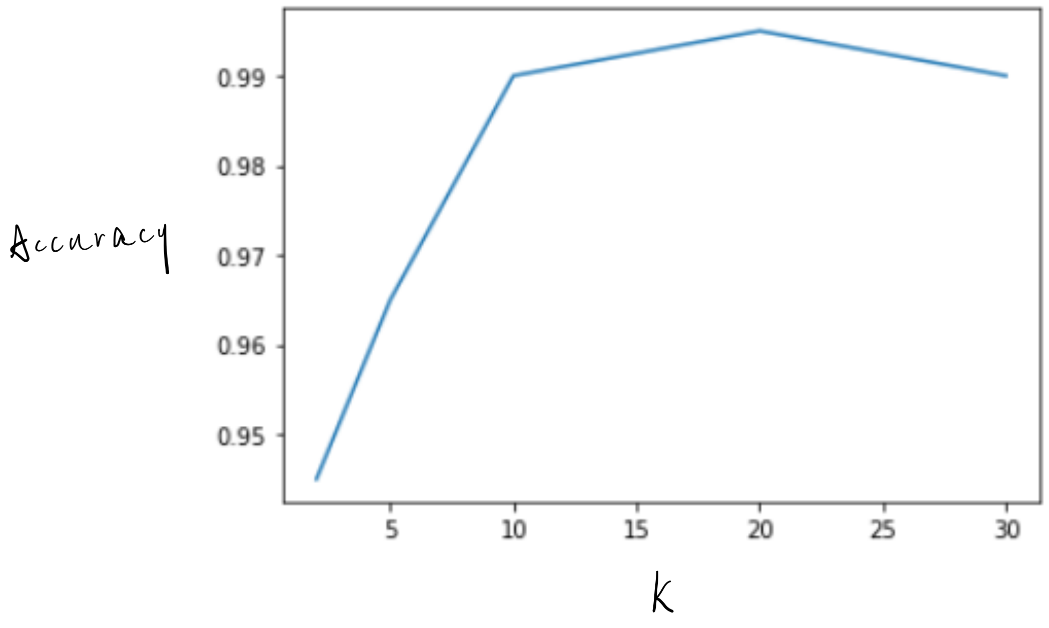
```python
def pca(x, k):
    """ PCA algorithm. Given the data matrix x and k,
    return the eigenvectors, mean of x, and the projected data (code vectors).

    Hint: You may use NumPy or SciPy to compute the eigenvectors/eigenvalues.

    :param x: A matrix with dimension N x D, where each row corresponds to
    one data point.
    :param k: int
        Number of dimension to reduce to.
    :return: Tuple of (Numpy array, Numpy array, Numpy array)
        WHERE
        v: A matrix of dimension D x k that stores top k eigenvectors
        mean: A vector of dimension D x 1 that represents the mean of x.
        proj_x: A matrix of dimension k x N where x is projected down to k dimension.
    """
    n, d = x.shape
    ######################################################################
    # TODO:                                                              #
    ######################################################################
    mean = x.mean(0).T
    c = x-mean.T
    sig = np.dot(c.T, c)/n
    v = lin.eig(sig)[1][:, 0:k]
    proj_x = np.dot(v.T, c.T)
    ######################################################################
    #                       END OF YOUR CODE                             #
    ######################################################################
    return v, mean, proj_x
```

3b)



```python
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 = pca(x_train, k)
        p_valid = np.full((x_valid.shape[0], 1), 0) # placeholder for the prediction of the validation set
        for i in range(x_valid.shape[0]):
            proj_v = np.dot(v.T, (x_valid[i, :] - mean).T)
            distance = np.sum((proj_x.T - proj_v.T)**2, axis=1) # distances from a validation point to each training points
            min_ind = np.argmin(distance)
            p_valid[i] = y_train[min_ind] # predict the validation set using 1-NN
        k = (p_valid == y_valid).sum() # find the number of correct predictions
        c = k/y_valid.shape[0]
        val_acc[j] = c
    #####################################################################
    #                       END OF YOUR CODE                            #
    #####################################################################
    plt.plot(k_lst, val_acc)
    plt.show()
```





3 c) I will choose K=20. Since on the plot for 2b), when K=20, the accuracy of the test reached a maximum

3d) My final test accuracy is 0.99 with K=20. In terms of the logistic model, in the way I set the hyperparameter: learning rate = 0.1, weight regularization = 0, iterations = 1000, the test accuracy is 0.97. So overall, the performance of applying PCA is better than the logistic regression.