Assignment 2

1.

1.1 $p(t=1|x) = \dfrac{p(x|t=1)p(t=1)}{p(x|t=1)p(t=1)+p(x|t=0)p(t=0)}$ (using Baye's Rule)

$= \dfrac{1}{1+\frac{p(x|t=0)p(t=0)}{p(x|t=1)p(t=1)}} = \dfrac{1}{1+\frac{\prod_{i=1}^{D}(2\pi\sigma_i^2)^{-\frac{1}{2}}\exp\left(\sum_{i=1}^{D}\frac{(x_i-\mu_{i0})^2}{-2\sigma_i^2}\right)(1-\alpha)}{\prod_{i=1}^{D}(2\pi\sigma_i^2)^{-\frac{1}{2}}\exp\left(\sum_{i=1}^{D}\frac{(x_i-\mu_{i1})^2}{-2\sigma_i^2}\right)\alpha}}$

$= \dfrac{1}{1+\exp\left(\sum_{i=1}^{D}\frac{(x_i-\mu_{i1})^2-(x_i-\mu_{i0})^2}{2\sigma_i^2}\right)\frac{1-\alpha}{\alpha}}$

$= \dfrac{1}{1+\exp\left(\sum_{i=1}^{D}\frac{2(\mu_{i0}-\mu_{i1})x_i+\mu_{i1}^2-\mu_{i0}^2}{2\sigma_i^2}+\ln\frac{1-\alpha}{\alpha}\right)}$

$= \dfrac{1}{1+\exp\left(\sum_{i=1}^{D}\frac{\mu_{i0}-\mu_{i1}}{\sigma_i^2}x_i+\sum_{i=1}^{D}\frac{\mu_{i1}^2-\mu_{i0}^2}{2\sigma_i^2}+\ln\frac{1-\alpha}{\alpha}\right)}$

$= \dfrac{1}{1+\exp\left(-\sum_{i=1}^{D}\frac{\mu_{i1}-\mu_{i0}}{\sigma_i^2}x_i-\left(\sum_{i=1}^{D}\frac{\mu_{i0}^2-\mu_{i1}^2}{2\sigma_i^2}+\ln\frac{\alpha}{1-\alpha}\right)\right)}$

Let $W_i = \dfrac{\mu_{i1}-\mu_{i0}}{\sigma_i^2}$, $b = \sum_{i=1}^{D}\dfrac{\mu_{i0}^2-\mu_{i1}^2}{2\sigma_i^2}+\ln\dfrac{\alpha}{1-\alpha}$

1.2 Let $\theta_i = x_i^T w + b$, $i \in [1, N]$.

$L = \prod_{i=1}^{N}\left(\frac{1}{1+e^{-\theta_i}}\right)^{t_i}\left(1-\frac{1}{1+e^{-\theta_i}}\right)^{1-t_i}$

$= \prod_{i=1}^{N}\left(\frac{1}{1+e^{-\theta_i}}\right)^{t_i}\left(\frac{1}{1+e^{\theta_i}}\right)^{1-t_i}$

$l = \log L = \sum_{i=1}^{N} t_i\log\frac{1}{1+e^{-\theta_i}}+(1-t_i)\log\frac{1}{1+e^{\theta_i}}$

$= \sum_{i=1}^{N} t_i\left(\log\frac{1}{1+e^{-\theta_i}}-\log\frac{1}{1+e^{\theta_i}}\right)+\log\frac{1}{1+e^{\theta_i}}$

$= \sum_{i=1}^{N} t_i\log\left(\frac{1+e^{\theta_i}}{1+e^{-\theta_i}}\right)+\log\frac{1}{1+e^{\theta_i}}$

$= \sum_{i=1}^{N} t_i\log e^{\theta_i}-\log(1+e^{\theta_i}) = \sum_{i=1}^{N} t_i\theta_i-\log(1+e^{\theta_i})$

$$l(w,b) = -l = \sum_{i=1}^{N} \log(1+e^{\theta_i}) - t_i\theta_i$$

$$\frac{dl}{dw_j} = \frac{dl}{d\theta_i}\frac{d\theta_i}{dw_j} = \left(\frac{1}{1+e^{-\theta_i}} - t_i\right) x_{ij},$$

write this as matrix:

$$\frac{dl}{dw} = X^T\left(\begin{pmatrix} \frac{1}{1+e^{-\theta_1}} \\ \frac{1}{1+e^{-\theta_2}} \\ \vdots \\ \frac{1}{1+e^{-\theta_N}} \end{pmatrix} - t\right)$$

$$\frac{dl}{db} = \begin{pmatrix} \frac{1}{1+e^{-\theta_1}} \\ \frac{1}{1+e^{-\theta_2}} \\ \vdots \\ \frac{1}{1+e^{-\theta_N}} \end{pmatrix} - t$$

1.3 $p(w,b|D,\lambda) = \frac{p(w,b,D|\lambda)}{p(D|\lambda)}$

$$\propto p(w,b,D|\lambda) = c\cdot p(w,b|\lambda)p(D|\lambda)$$

$$= c\cdot\prod_{i=1}^{N}\left(\frac{1}{1+e^{-x_i w - b}}\right)^{t_i}\left(1-\frac{1}{1+e^{-x_i w - b}}\right)^{1-t_i}\left(2\pi\frac{1}{\lambda}\right)^{-\frac{N}{2}} exp\left(-\frac{\lambda}{2}w^Tw\right)$$

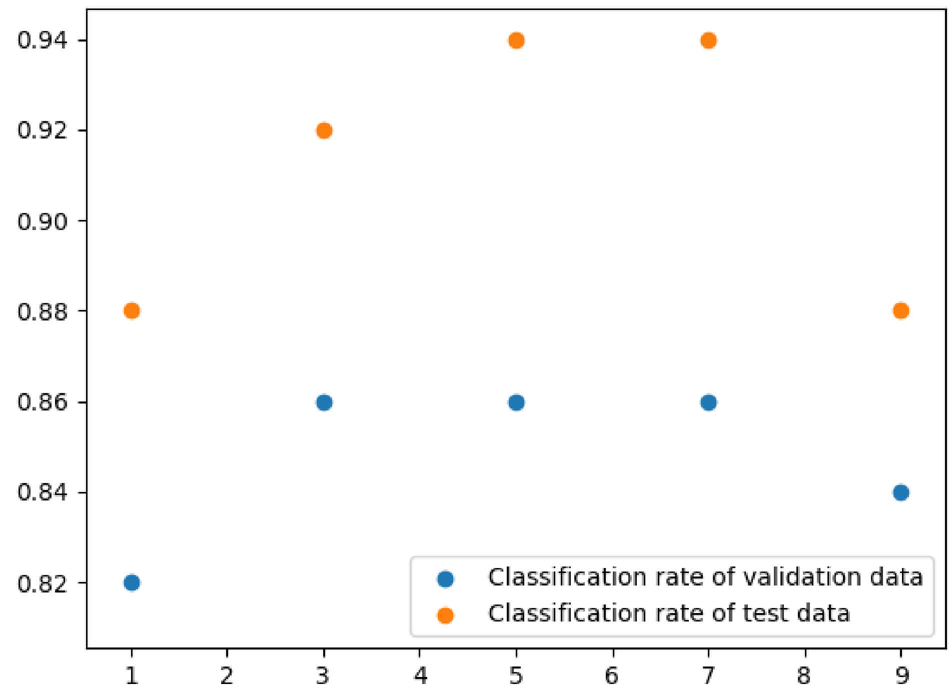
$$= c_1\prod_{i=1}^{N}\left(\frac{1}{1+e^{-x_i w - b}}\right)^{t_i}\left(1-\frac{1}{1+e^{-x_i w - b}}\right)^{1-t_i} exp\left(-\frac{\lambda}{2}w^Tw\right)$$

$$l_{post}(w,b) = -\left(\log c_1 - l(w,b) - \frac{\lambda}{2}w^Tw\right)$$

$$= l(w,b) + \frac{\lambda}{2}w^Tw - \log c_1$$

$$= l(w,b) + \frac{\lambda}{2}\sum_{i=1}^{D} w_i^2 + c_2$$

2.1



The classification rate on validation set has the trend of increasing then decreasing. We choose k = 5 according to the classification rate performance on validation set. K = 3 and k = 7 both have classification rate 86% as high as k = 5. The test performance corresponds to validation set generally.

Code of 2.1

```
import numpy as np
import matplotlib.pyplot as plt
from run_knn import run_knn
from logistic_regression_template import run_logistic_regression

# (a)
train = np.load("mnist_train.npz")
# print(train.files)
train_data = train['train_inputs']
# print(train_data.shape)
train_labels = train['train_targets']

valid = np.load("mnist_valid.npz")
# print(valid.files)
valid_data = valid['valid_inputs']
# print(valid_data.shape)
valid_labels = valid['valid_targets']

test = np.load("mnist_test.npz")
# print(test.files)
test_data = test['test_inputs']
# print(test_data.shape)
test_labels = test['test_targets']


classification_rate_validation = []
classification_rate_test = []
for k in [1, 3, 5, 7, 9]:
```

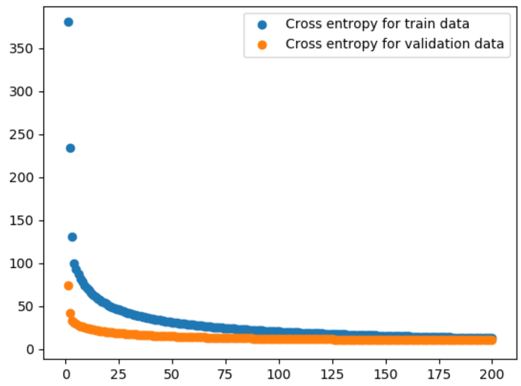
A2_Q2a.py | logistic_regression_template.py | logistic.py

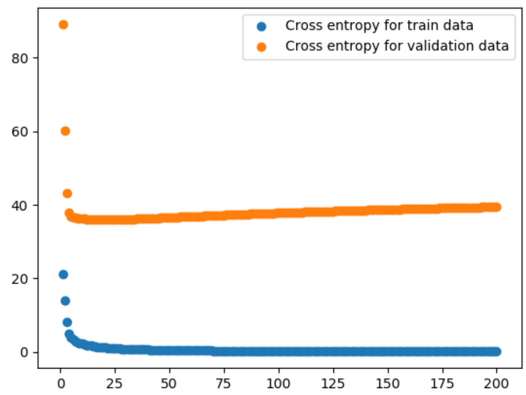
```python
# print(test_data.shape)
test_labels = test['test_targets']


classification_rate_validation = []
classification_rate_test = []
for k in [1, 3, 5, 7, 9]:
    vk_labels = run_knn(k, train_data, train_labels, valid_data)
    tk_labels = run_knn(k, train_data, train_labels, test_data)
    classification_rate_validation.append(np.count_nonzero(vk_labels == valid_labels) / len(val
    classification_rate_test.append(np.count_nonzero(tk_labels == test_labels) / len(test_label

print(classification_rate_validation)
print(classification_rate_test)

plt.scatter(np.array([1, 3, 5, 7, 9]), classification_rate_validation)
plt.scatter(np.array([1, 3, 5, 7, 9]), classification_rate_test)
plt.legend(['Classification rate of validation data', 'Classification rate of test data'])
plt.show()
```

2.2

The best hyperparameter I found for this model is learning rate = 0.1, num_interations = 200. Final cross entropy and classification error on the training, validation and test sets are:

```
TRAIN CE:12.710649916810699 TRAIN FRAC:100.0  VALID CE:10.610897864366496  VALID FRAC:88.0
TEST CE:10.452382270510103 TEST FRAC:92.0
```





The left plot is the cross entropy change for mnist_train set and validation set. The right pot is the cross entropy change for minst_train_small set and validation set. With the best hyperparameter set I choose, the two plot remains relatively stable in each time I run the code.

Code for 2.2

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

    Note: N is the number of examples and
          M is the number of features per example.

    Inputs:
        weights:    (M+1) x 1 vector of weights, where the last element
                    corresponds to the bias (intercepts).
        data:       N x M data matrix where each row corresponds
                    to one data point.
    Outputs:
        y:          :N x 1 vector of probabilities. This is the output of the classifier.
    """
    intercept = np.ones((data.shape[0], 1))
    x = np.append(data, intercept, axis=1)
    log_odds = np.dot(x, weights)
    y = sigmoid(log_odds)
    return y


def evaluate(targets, y):
    """
    Compute evaluation metrics.
    Inputs:
        targets : N x 1 vector of targets.
```

nn.py | A2_Q2a.py | A2_Q2c.py | logistic_regression_template.py | logistic.py

```python
        targets : N x 1 vector of targets.
        y       : N x 1 vector of probabilities.
    Outputs:
        ce           : (scalar) Cross entropy. CE(p, q) = E_p[-log q]. Here we want to compute
        frac_correct : (scalar) Fraction of inputs classified correctly.
    """
    ce = -(np.dot(targets.T, np.log(y)) + np.dot((1 - targets).T, np.log(1 - y)))
    y = (y >= 0.5).astype(int)
    frac_correct = np.count_nonzero(y == targets) / len(targets)
    return ce, frac_correct


def logistic(weights, data, targets, hyperparameters):
    """
    Calculate negative log likelihood and its derivatives with respect to weights.
    Also return the predictions.

    Note: N is the number of examples and
          M is the number of features per example.

    Inputs:
        weights:    (M+1) x 1 vector of weights, where the last element
                    corresponds to bias (intercepts).
        data:       N x M data matrix where each row corresponds
                    to one data point.
        targets:    N x 1 vector of targets class probabilities.
        hyperparameters: The hyperparameters dictionary.
```

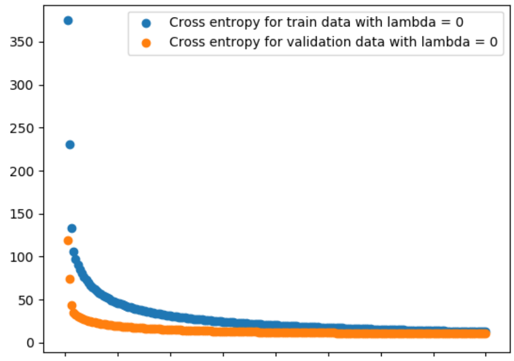
43:1 LF UTF-8

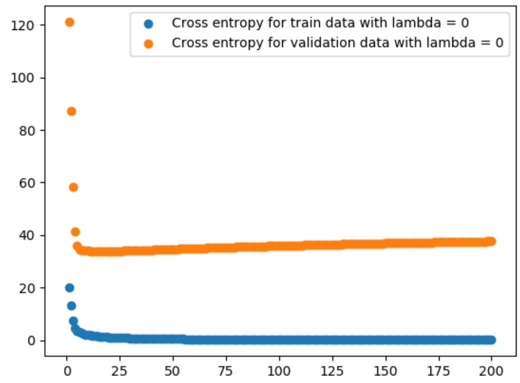
```python
        M is the number of features per example.

    Inputs:
        weights:    (M+1) x 1 vector of weights, where the last element
                    corresponds to bias (intercepts).
        data:       N x M data matrix where each row corresponds
                    to one data point.
        targets:    N x 1 vector of targets class probabilities.
        hyperparameters: The hyperparameters dictionary.

    Outputs:
        f:        The sum of the loss over all data points. This is the objective that we want
        df:       (M+1) x 1 vector of derivative of f w.r.t. weights.
        y:        N x 1 vector of probabilities.
    """

    y = logistic_predict(weights, data)
    intercept = np.ones((data.shape[0], 1))
    x = np.append(data, intercept, axis=1)
    log_odds = np.dot(x, weights)
    f = np.sum(np.log(1+np.exp(log_odds)) - np.multiply(log_odds, targets))
    df = np.dot(x.T, y - targets)
    return f, df, y


def logistic_pen(weights, data, targets, hyperparameters):
    """
    Calculate negative log likelihood and its derivatives with respect to weights.
    Also return the predictions
```

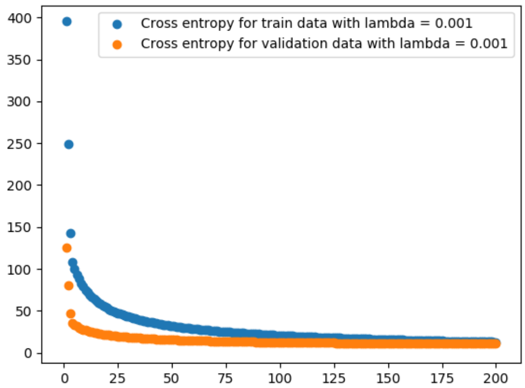
```python
        # print some stats
        # print("ITERATION:{}  TRAIN NLOGL:{}  TRAIN CE:{} "
        #       "TRAIN FRAC:{}  VALID NLOGL:{}  VALID CE:{}  VALID FRAC:{}".format(
        #     t + 1, f / N, cross_entropy_train, frac_correct_train * 100, f_valid / valid_inp
        #     cross_entropy_valid, frac_correct_valid * 100))
        print("ITERATION:{}  TEST NLOGL:{}  TEST CE:{} "
              "TEST FRAC:{}".format(
            t + 1, f_test[0][0] / test_targets.shape[0],
            cross_entropy_test[0][0], frac_correct_test * 100))

    plt.scatter(np.arange(1, hyperparameters['num_iterations'] + 1), TRAIN_CE)
    plt.scatter(np.arange(1, hyperparameters['num_iterations'] + 1), VALID_CE)
    plt.legend(['Cross entropy for train data with lambda = ' + str(lmbda),
                'Cross entropy for validation data with lambda = ' + str(lmbda)])
    plt.show()
```

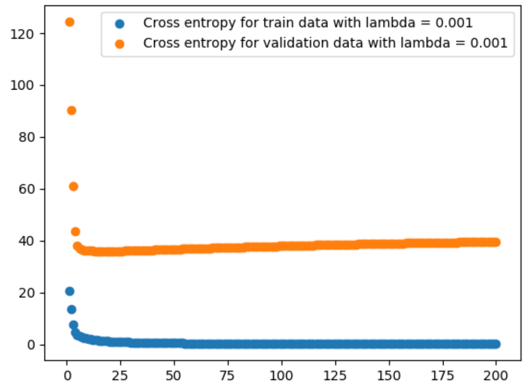
2.3

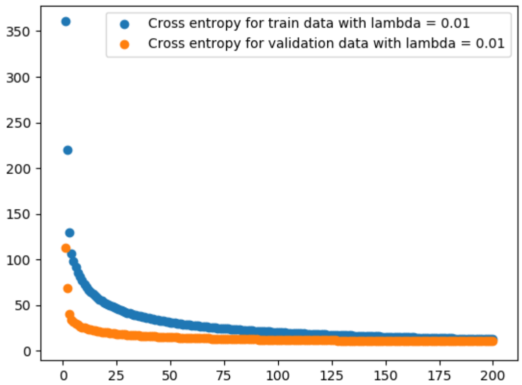
Here are several plots:

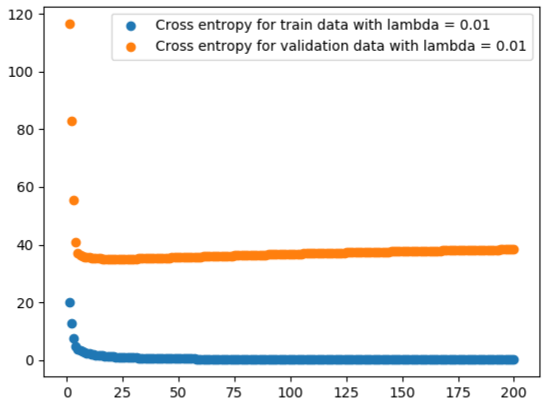

Cross entropy for train data with lambda = 0
Cross entropy for validation data with lambda = 0

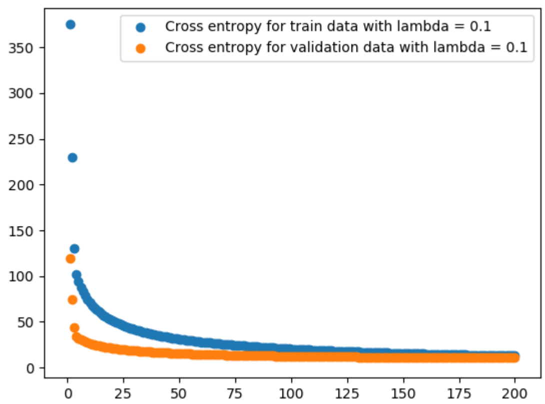

Cross entropy for train data with lambda = 0
Cross entropy for validation data with lambda = 0

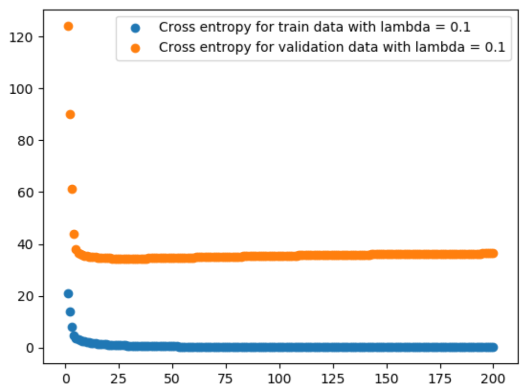

Cross entropy for train data with lambda = 0.001
Cross entropy for validation data with lambda = 0.001

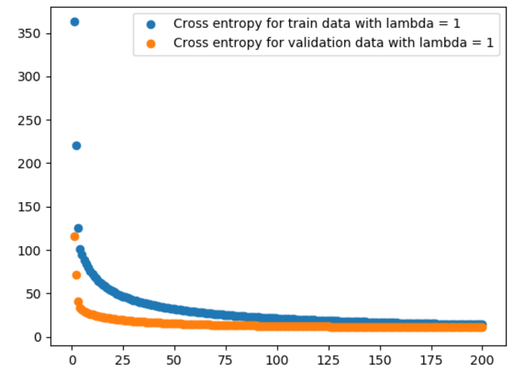

Cross entropy for train data with lambda = 0.001
Cross entropy for validation data with lambda = 0.001

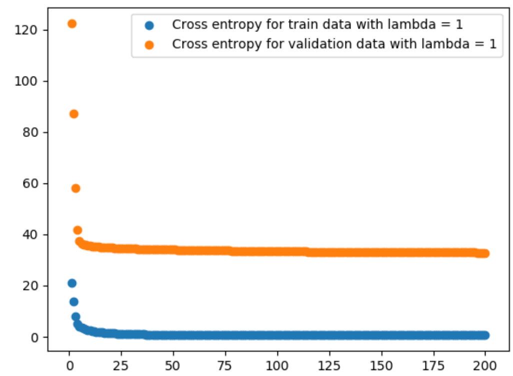

Cross entropy for train data with lambda = 0.01
Cross entropy for validation data with lambda = 0.01

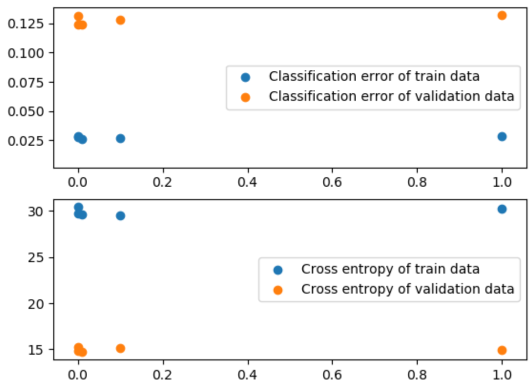

Cross entropy for train data with lambda = 0.01
Cross entropy for validation data with lambda = 0.01

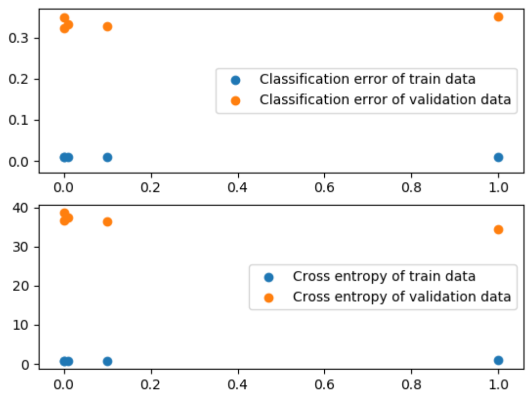

Cross entropy for train data with lambda = 0.1
Cross entropy for validation data with lambda = 0.1


Cross entropy for train data with lambda = 0.1
Cross entropy for validation data with lambda = 0.1









The plots on the left are cross entropy changes each iteration of train and validation set with different λ using training set minst_train, plots on the right are generated in the same procedure as left plots but using training set minst_small.

The last two plots both containing 2 subplots representing the average change over iterations of cross entropy and classification error change against different values of λ. Both cross entropy and classification error have the trend of increasing first then decreasing. This is reasonable as when we increase the penalty λ , we may fix some collinearity issue in the input set. However, when the penalty is large, our estimator could be severely biased, so that classification error will increase. We choose λ = 0.001 at last, the test classification error is 8%, with 92% accuracy.

```
ITERATION:200   TEST NLOGL:0.20692403247235058   TEST CE:10.341793538031553 TEST FRAC:92.0
```

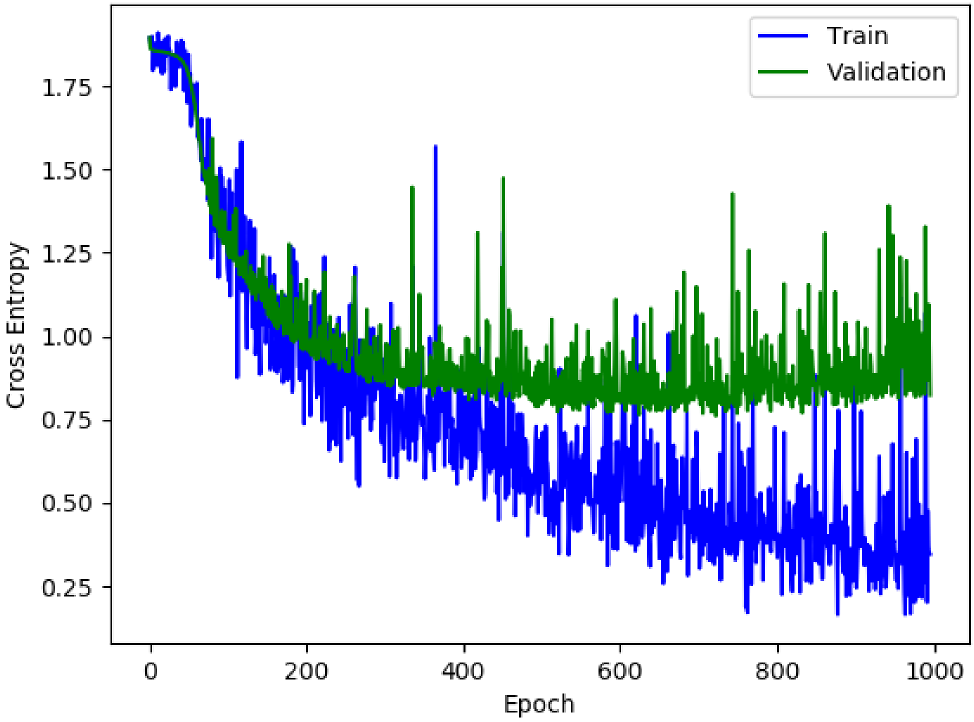
Finally, we see that there is no significant difference in adding the penalty. We think this is probably because the data values in 28*28 pixels are kind of random distributed, so columns in input matrix are quite linearly independent so that not much regularization is needed to modify this model.

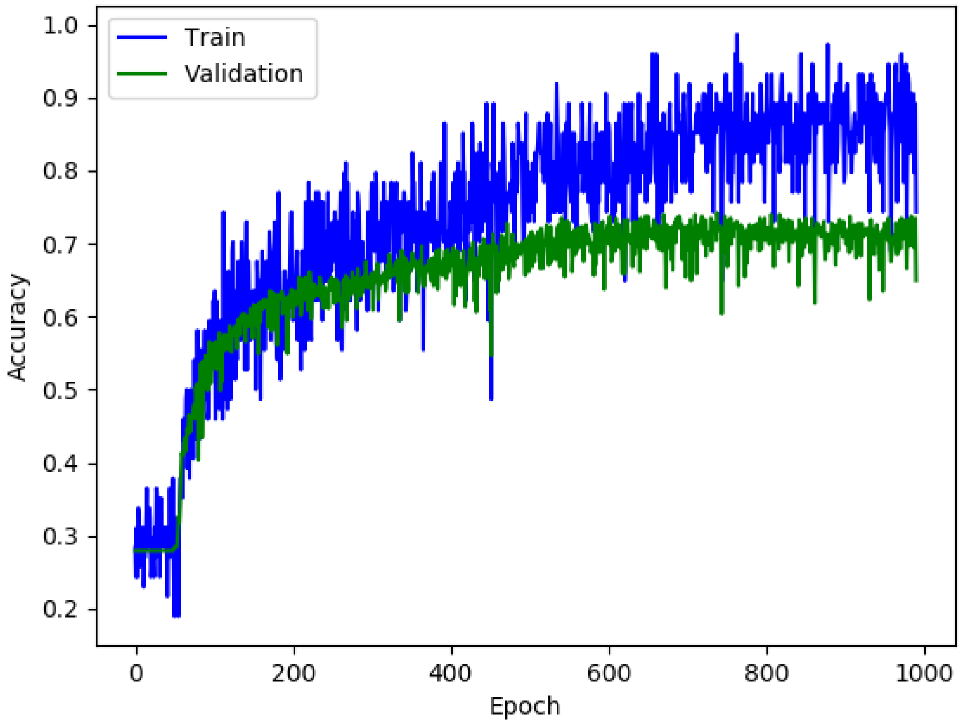
Code for 2.3:

```
from logistic_regression_template import *
import matplotlib.pyplot as plt

list_train_error_avg = []
list_train_ce_avg = []
list_valid_error_avg = []
list_valid_ce_avg = []
list_lmbda = [0, 0.001, 0.01, 0.1, 1]
for lmbda in list_lmbda:
    train_error_avg, train_ce_avg,\
        valid_error_avg, valid_ce_avg = run_pen_logistic_regression(lmbda)
    list_train_error_avg.append(train_error_avg)
    list_train_ce_avg.append(train_ce_avg)
    list_valid_error_avg.append(valid_error_avg)
    list_valid_ce_avg.append(valid_ce_avg)

fig, (ax1, ax2) = plt.subplots(2)
ax1.scatter(np.array(list_lmbda), list_train_error_avg)
ax1.scatter(np.array(list_lmbda), list_valid_error_avg)
ax1.legend(['Classification error of train data', 'Classification error of validation data'])
ax2.scatter(np.array(list_lmbda), list_train_ce_avg)
ax2.scatter(np.array(list_lmbda), list_valid_ce_avg)
ax2.legend(['Cross entropy of train data', 'Cross entropy of validation data'])
plt.show()

run_pen_logistic_regression(0.001)
```

3.1

We get the two plots indicating cross entropy and accuracy with random hyperparameters:

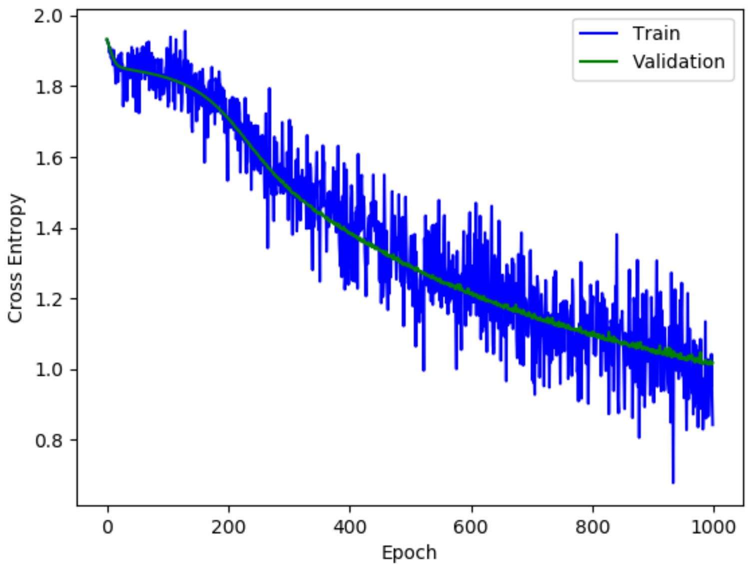


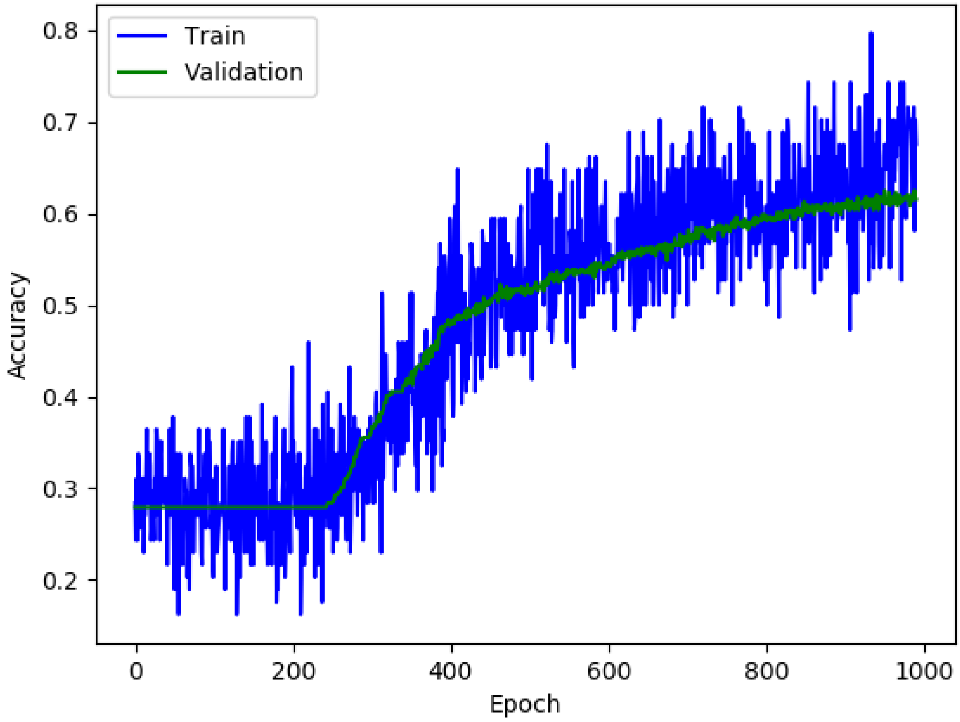


Both training and validation set has the same trend: increasing accuracy, decreasing cross entropy.

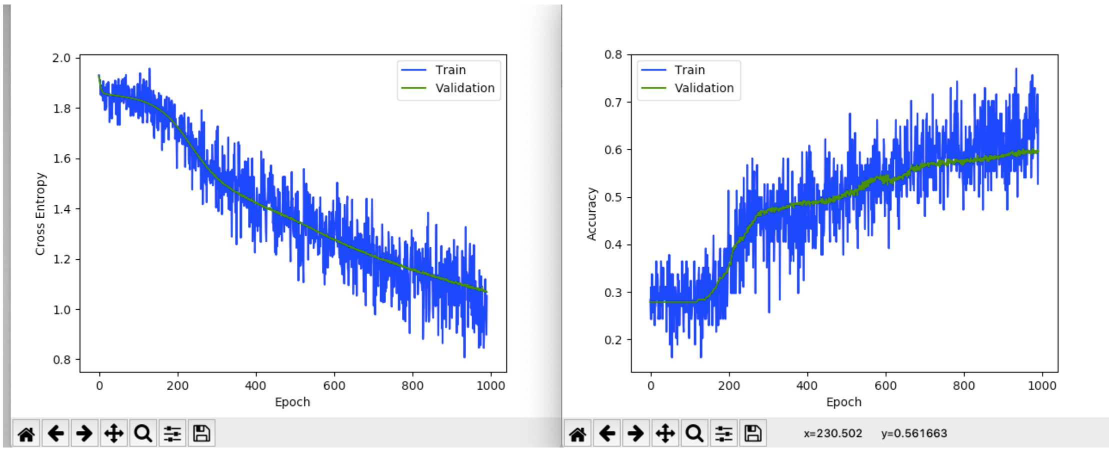
3.2

After experimenting, we find that as learning and increases, accuracy increases. Plot for learning rate = 0.001:





It seems that momentum = 0.6 works best:



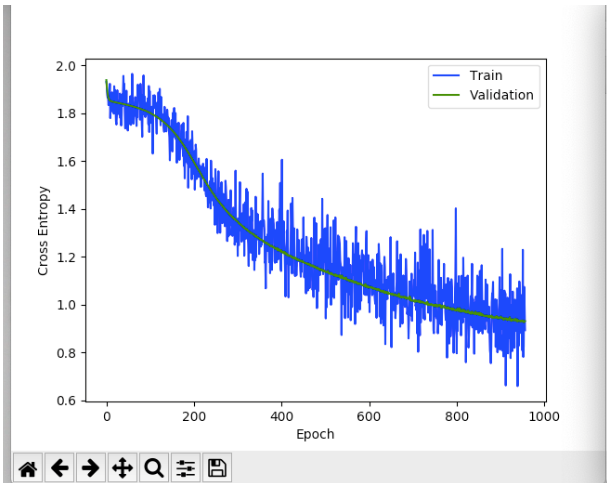
Statistics for the above plots:
CE: Train 1.03181 Validation 1.06957 Test 1.09550
Acc: Train 0.62893 Validation 0.59189 Test 0.59740

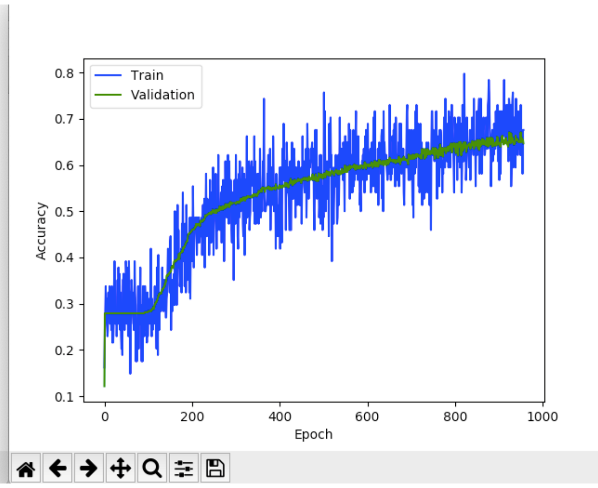
We found that batch size increases, the accuracy decreases and the cross entropy increases. Therefore, we choose batch size = 100.

3.3

Due to time pressure, we only tested a few choices for number of hidden layers and this choice seems to work better. Thus, we guess that with increasing hidden layers, the accuracy should increase, but should be a upper bound for this relation.

HIDDEN [32,64]:





Statistics for the above plots:
CE: Train 0.98503 Validation 1.05586 Test 1.05219
Acc: Train 0.64878 Validation 0.60382 Test 0.60519

3.4
Some faces may have ambiguous face expressions, so it may be hard to classify them into the 7 categories. We try to show those face photos from the original data set.

Code for 3.1-3.4:

```
def AffineBackward(grad_y, h, w):
    """Computes gradients of affine transformation.
    hint: you may need the matrix transpose np.dot(A,B).T = np.dot(B,A) and (A.T).T = A

    Args:
        grad_y: gradient from last layer
        h: inputs from the hidden layer
        w: weights

    Returns:
        grad_h: Gradients wrt. the inputs/hidden layer.
        grad_w: Gradients wrt. the weights.
        grad_b: Gradients wrt. the biases.
    """
    ###########################
    # Insert your code here.
    grad_w = np.dot(h.T, grad_y)
    grad_h = np.dot(grad_y, w.T)
    grad_b = np.sum(grad_y, axis=0).T
    return grad_h, grad_w, grad_b
    ###########################
```

```python

def ReLUBackward(grad_h, z):
    """Computes gradients of the ReLU activation function wrt. the unactivated inputs.

    Returns:
        grad_z: Gradients wrt. the hidden state prior to activation.
    """
    ###########################
    # Insert your code here.
    grad_z = np.zeros(grad_h.shape)
    for i in range(grad_h.shape[0]):
        for j in range(grad_h.shape[1]):
            if z[i][j] > 0:
                grad_z[i][j] = grad_h[i][j]
    return grad_z
    ###########################

```

```python
def NNUpdate(model, eps, momentum):
    """Update NN weights.

    Args:
        model:    Dictionary of all the weights.
        eps:      Learning rate.
        momentum: Momentum.
    """
    ###########################
    # Insert your code here.
    # Update the weights.
    model['W1'] = model['W1'] - eps * (momentum * model['W1'] +
                                       (1 - momentum) * model['dE_dW1'])
    model['W2'] = model['W2'] - eps * (momentum * model['W2'] +
                                       (1 - momentum) * model['dE_dW2'])
    model['W3'] = model['W3'] - eps * (momentum * model['W3'] +
                                       (1 - momentum) * model['dE_dW3'])
    model['b1'] = model['b1'] - eps * (momentum * model['b1'] +
                                       (1 - momentum) * model['dE_db1'])
    model['b2'] = model['b2'] - eps * (momentum * model['b2'] +
                                       (1 - momentum) * model['dE_db2'])
    model['b3'] = model['b3'] - eps * (momentum * model['b3'] +
                                       (1 - momentum) * model['dE_db3'])
    ###########################
```