CSC311 HW2

Ziqi Chen #1002576722

1. Logistic Regression

a. Bayes' Rule

Based on Bayes' Rule

$$p(t=1|x) = \frac{p(x|t=1) * p(t=1)}{p(x)}$$

$$p(t=1|x) = \frac{p(x|t=1)p(t=1)}{p(x|t=1)p(t=1) + p(x|t=0)p(t=0)}$$

$$p(t=1|x) = \frac{1}{1 + \frac{p(x|t=0)p(t=0)}{p(x|t=1)p(t=1)}}$$

We know:

$p(x_i|t=k)p(t=k) \sim N(\mu_{ik}, {\sigma_i}^2)$

And thus:

$$p(X|t=k) = \prod_{i=1}^{D} \frac{1}{\sqrt{2\pi{\sigma_i}^2}} exp\,(-\frac{(x_i - \mu_{ik})^2}{2{\sigma_i}^2})$$

So:

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

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

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

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

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

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

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

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

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

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

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

Therefore we have

$$w = (\dots w_i \dots)^T$$

$$w_i = -\frac{\mu_{i0} - \mu_{i1}}{{\sigma_i}^2}$$

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

b. Maximum Likelihood Estimate

We have:

$$P\left(t^{(n)}=1\middle|x^{(n)}\right)=\sigma(w^Tx^{(n)}+b)$$
$$P\left(t^{(n)}=0\middle|x^{(n)}\right)=1-\sigma\left(w^Tx^{(n)}+b\right)$$

Combine these two:

$$P\left(t^{(n)}\middle|x^{(n)}\right)=(\sigma(w^Tx^{(n)}+b))^{t^{(n)}}(1-\sigma\left(w^Tx^{(n)}+b\right))^{(1-t^{(n)})}$$

Therefore:

$$likelihood=\prod_{i=1}^{n}(\sigma(w^Tx+b))^{t_i}(1-\sigma(w^Tx+b))^{(1-t_i)}$$
$$negative\ loglikellihood\ L(w,b)=-\log(likelihood)$$
$$L(w,b)=-\sum_{j=1}^{n}(t_j\log(\sigma\left(w^Tx_j+b\right))+(1-t_j)\log(1-\sigma\left(w^Tx_j+b\right)))$$

Substitute with $\sigma\left(w^Tx_j+b\right))=\frac{1}{1+\exp\left(-\sum_{i=1}^{D}w_ix_{ij}-b\right)}$

$$L(w,b)=-\sum_{j=1}^{n}(t_j\log(\frac{1}{1+\exp\left(-\sum_{i=1}^{D}w_ix_{ij}-b\right)})+(1-t_j)\log(1-\frac{1}{1+\exp\left(-\sum_{i=1}^{D}w_ix_{ij}-b\right)}))$$

Derivative:

$$\frac{\partial L(w,b)}{\partial w}=\sum_{j=1}^{n}(t_j\left(1+\exp\left(-\sum_{i=1}^{D}w_ix_{ij}-b\right)\right)\exp\left(-\sum_{i=1}^{D}w_ix_{ij}-b\right)\sum_{i=1}^{D}x_{ij}+$$
$$(1-t_j)(\frac{1}{\exp\left(-\sum_{i=1}^{D}w_ix_{ij}-b\right))}+1)\ \exp\left(-\sum_{i=1}^{D}w_ix_{ij}-b\right)\sum_{i=1}^{D}x_{ij})$$
$$\frac{\partial L(w,b)}{\partial b}=\sum_{j=1}^{n}(t_j\left(1+\exp\left(-\sum_{i=1}^{D}w_ix_{ij}-b\right)\right)\exp\left(-\sum_{i=1}^{D}w_ix_{ij}-b\right)+$$
$$(1-t_j)(\frac{1}{\exp\left(-\sum_{i=1}^{D}w_ix_{ij}-b\right))}+1)\ \exp\left(-\sum_{i=1}^{D}w_ix_{ij}-b\right))$$

c. L2 Regularization

We have

$$L(w,b)=-\sum_{j=1}^{n}(t_j\log(\sigma\left(w^Tx_j+b\right))+(1-t_j)\log(1-\sigma\left(w^Tx_j+b\right)))$$

Now:

$$expression=\alpha p(w,b|D)\propto\alpha\ p(D|w,b)p(w,b)$$

And since p(D|w,b) is sum so we don't need to look at it.

And then

$$likelihood\propto\prod_{i=1}^{D}(\sigma(w^Tx+b))^{t_i}(1-\sigma(w^Tx+b))^{(1-t_i)}\,\alpha\,N(w_i|0,\frac{1}{\lambda})$$

$$L_{post}(w,b)=-\sum_{i=1}^{D}(t_j\log(\sigma\left(w^Tx_j+b\right))+(1-t_j)\log\left(1-\sigma\left(w^Tx_j+b\right)\right)+\alpha\frac{\lambda}{2}{w_i}^2+\frac{1}{2}\log\left(\frac{2\pi}{\lambda}\right))$$

$$L_{post}(w,b)=-\sum_{i=1}^{D}(t_j\log\left(\sigma\left(w^Tx_j+b\right)\right)+(1-t_j)\log\left(1-\sigma\left(w^Tx_j+b\right)\right))-\sum_{i=1}^{D}\left(\alpha\frac{\lambda}{2}{w_i}^2\right)-\sum_{i=1}^{D}(\frac{1}{2}\log\left(\frac{2\pi}{\lambda}\right))$$

$$L_{post}(w,b)=-\sum_{i=1}^{D}(t_j\log\left(\sigma\left(w^Tx_j+b\right)\right)+(1-t_j)\log\left(1-\sigma\left(w^Tx_j+b\right)\right))-\alpha\frac{\lambda}{2}\sum_{i=1}^{D}({w_i}^2)-\frac{D}{2}\log\left(\frac{2\pi}{\lambda}\right)$$

$$and\ as\ we\ recognize,-\sum_{i=1}^{D}(t_j\log\left(\sigma\left(w^Tx_j+b\right)\right)+(1-t_j)\log\left(1-\sigma\left(w^Tx_j+b\right)\right))=L(w,b)$$

$$\frac{D}{2}\log\left(\frac{2\pi}{\lambda}\right)=Constant$$

$$and\ thus\ -\alpha\frac{\lambda}{2}\sum_{i=1}^{D}({w_i}^2)=\frac{\lambda}{2}\sum_{i=1}^{D}({w_i}^2)$$

$Therefore\ \alpha=-1$

$and\ our\ expression\ would\ be$ −p(w,b)

$$Then\ L_{post}(w,b)=L(w,b)+\frac{\lambda}{2}\sum_{i=1}^{D}({w_i}^2)\ +Constant$$

$$\frac{\partial L_{post}(w,b)}{\partial w}=\frac{\partial L(w,b)}{\partial w}+\lambda\sum_{i=1}^{D}w_i$$

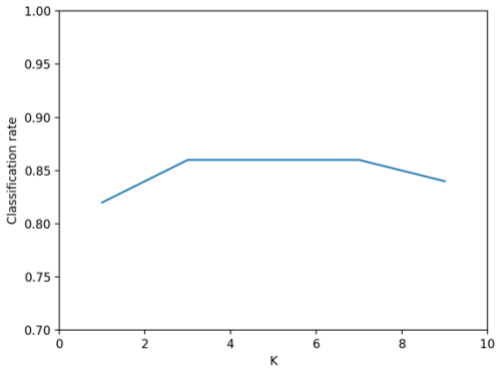
$$\frac{\partial L_{post}(w,b)}{\partial b}=\frac{\partial L(w,b)}{\partial b}$$

Both derivative $\frac{\partial L(w,b)}{\partial w}\ and\ \frac{\partial L(w,b)}{\partial b}$ are derived in 1.(b)

2. Logistic Regression vs. KNN

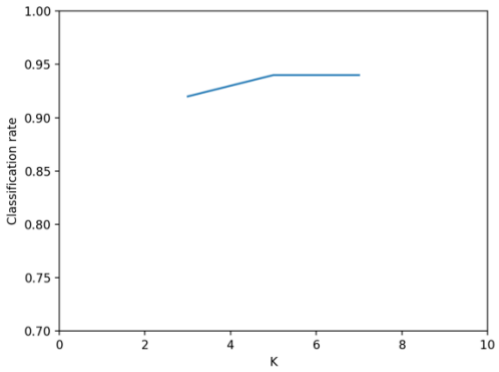
a. KNN

Result on validation set:



We choose k^* = 5, then plot 3, 5, 7

Result on test data:



As we can see the test result is better than validation result

This is happening maybe because test set has a lot case that's more similar to training set, whereas validation set does not.

Code:

```
def evaluate(y, target):
    correct = zip(*np.where(target==y))
    return len(correct) * 1.0/len(target)
```

```
def q21_script1():
    train_data, train_labels = utils.load_train()
    valid_data, valid_labels = utils.load_valid()
    k_list = [1,3,5,7,9]
    result = []
    for k in k_list:
        y = rk.run_knn(k, train_data, train_labels, valid_data)
        score = evaluate(y, valid_labels)
        result.append(score)
    result = np.array(result)
    plt.plot(k_list, result)
    plt.xlabel('K')
    plt.ylabel('Classification rate')
    plt.axis([0,10,.70,1.0])
    plt.show()
```

b. Selection of hyperparameters: learning rate = 0.005, num_iterations = 1500

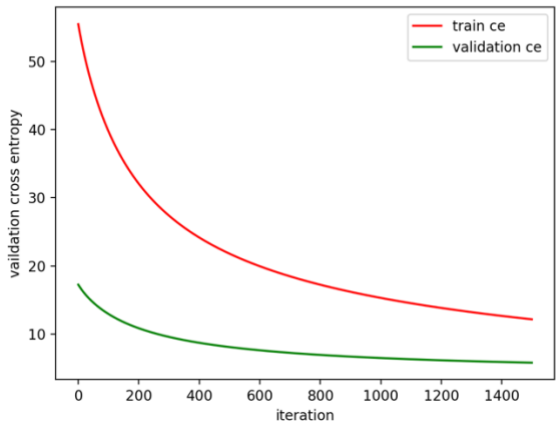
Final VALID FRAC = 90.00 for train data

```
ITERATION:1485  TRAIN NLOGL:0.16  TRAIN CE:12.246049 TRAIN FRAC:99.38  VALID CE:5.827228  VALID FRAC:90.00
ITERATION:1486  TRAIN NLOGL:0.16  TRAIN CE:12.241202 TRAIN FRAC:99.38  VALID CE:5.826239  VALID FRAC:90.00
ITERATION:1487  TRAIN NLOGL:0.16  TRAIN CE:12.236359 TRAIN FRAC:99.38  VALID CE:5.825252  VALID FRAC:90.00
ITERATION:1488  TRAIN NLOGL:0.16  TRAIN CE:12.231521 TRAIN FRAC:99.38  VALID CE:5.824266  VALID FRAC:90.00
ITERATION:1489  TRAIN NLOGL:0.16  TRAIN CE:12.226688 TRAIN FRAC:99.38  VALID CE:5.823281  VALID FRAC:90.00
ITERATION:1490  TRAIN NLOGL:0.16  TRAIN CE:12.221859 TRAIN FRAC:99.38  VALID CE:5.822297  VALID FRAC:90.00
ITERATION:1491  TRAIN NLOGL:0.16  TRAIN CE:12.217034 TRAIN FRAC:99.38  VALID CE:5.821314  VALID FRAC:90.00
ITERATION:1492  TRAIN NLOGL:0.16  TRAIN CE:12.212215 TRAIN FRAC:99.38  VALID CE:5.820332  VALID FRAC:90.00
ITERATION:1493  TRAIN NLOGL:0.16  TRAIN CE:12.207399 TRAIN FRAC:99.38  VALID CE:5.819352  VALID FRAC:90.00
ITERATION:1494  TRAIN NLOGL:0.16  TRAIN CE:12.202588 TRAIN FRAC:99.38  VALID CE:5.818373  VALID FRAC:90.00
ITERATION:1495  TRAIN NLOGL:0.16  TRAIN CE:12.197781 TRAIN FRAC:99.38  VALID CE:5.817395  VALID FRAC:90.00
ITERATION:1496  TRAIN NLOGL:0.16  TRAIN CE:12.192979 TRAIN FRAC:99.38  VALID CE:5.816418  VALID FRAC:90.00
ITERATION:1497  TRAIN NLOGL:0.16  TRAIN CE:12.188182 TRAIN FRAC:99.38  VALID CE:5.815442  VALID FRAC:90.00
ITERATION:1498  TRAIN NLOGL:0.16  TRAIN CE:12.183389 TRAIN FRAC:99.38  VALID CE:5.814467  VALID FRAC:90.00
ITERATION:1499  TRAIN NLOGL:0.16  TRAIN CE:12.178600 TRAIN FRAC:99.38  VALID CE:5.813493  VALID FRAC:90.00
ITERATION:1500  TRAIN NLOGL:0.16  TRAIN CE:12.173816 TRAIN FRAC:99.38  VALID CE:5.812521  VALID FRAC:90.00
```

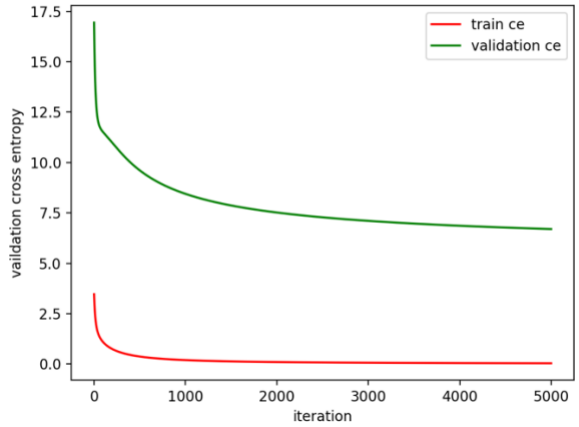
And final VALID FRAC = 70 for train_small data, but really small train ce, learning rate = 0.005, num_iterations = 5000

```
ITERATION:4996  TRAIN NLOGL:0.01  TRAIN CE:0.038898 TRAIN FRAC:100.00  VALID CE:6.699652  VALID FRAC:70.00
ITERATION:4997  TRAIN NLOGL:0.01  TRAIN CE:0.038890 TRAIN FRAC:100.00  VALID CE:6.699514  VALID FRAC:70.00
ITERATION:4998  TRAIN NLOGL:0.01  TRAIN CE:0.038882 TRAIN FRAC:100.00  VALID CE:6.699377  VALID FRAC:70.00
ITERATION:4999  TRAIN NLOGL:0.01  TRAIN CE:0.038874 TRAIN FRAC:100.00  VALID CE:6.699240  VALID FRAC:70.00
ITERATION:5000  TRAIN NLOGL:0.01  TRAIN CE:0.038866 TRAIN FRAC:100.00  VALID CE:6.699102  VALID FRAC:70.00
```

For train data:



For train_small:



Code:

In [2]:
```
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.
    """

    N, M = data.shape[0], data.shape[1]
    y = np.zeros((N,1))
    new_data=np.hstack((data, np.ones((N,1))))
    y = np.dot( new_data, weights)
    y = sigmoid(y)
    return y
```

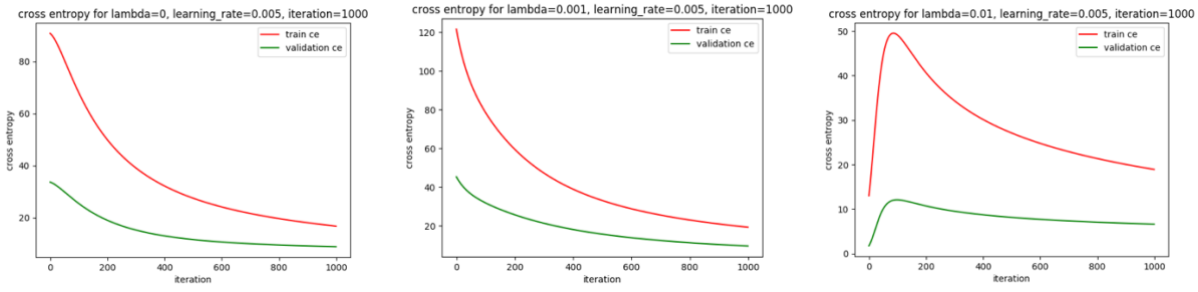
In [3]:
```
def evaluate(targets, y):
    """
    Compute evaluation metrics.
    Inputs:
        targets : N x 1 vector of targets.
        y       : N x 1 vector of probabilities.
    Outputs:
        ce           : (scalar) Cross entropy. CE(p, q) = E_p[-log q]. Here we want to compute CE(targets, y)
        frac_correct : (scalar) Fraction of inputs classified correctly.
    """
    ce = - np.sum(np.dot(targets.T, np.log(y)))
    correct_prediction = len(zip(*np.where((y > 0.5) & (targets == 1)))) + len(zip(*np.where((y<0.5) & (targets==0))))
    frac_correct = float(correct_prediction) / len(targets)
    return ce, frac_correct
```

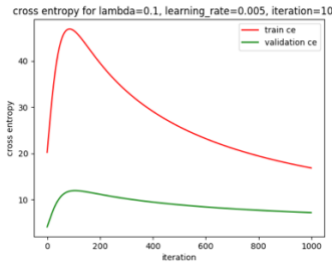
In [4]:
```
def logistic(weights, data, targets, hyperparameters):
    """
    Calculate negative log likelihood and its derivatives with respect to weights.
    Also return the predictions.

    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.

    Outputs:
        f:       The sum of the loss over all data points. This is the objective that we want to minimize.
        df:      (M+1) x 1 vector of derivative of f w.r.t. weights.
        y:       N x 1 vector of probabilities.
    """
    N, M = data.shape[0], data.shape[1]
    y = logistic_predict(weights, data)
    f = np.sum(-np.dot(targets.T,np.log(y))) - np.sum(np.dot((1-targets.T), np.log(1 - y)))
    new_data = np.hstack((data, np.ones((N,1))))
    df = np.dot(new_data.T, (y - targets))
    return f, df, y
```

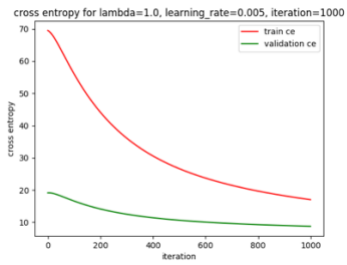
c. Penalized Logistic Regression

We are doing learning rate = 0.005, and 1000 iterations

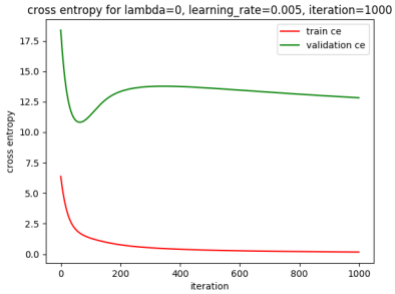
Result For Train data:

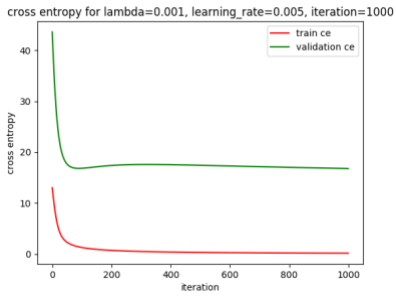


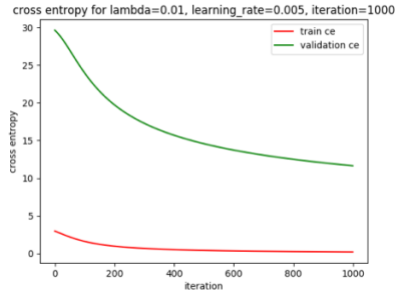


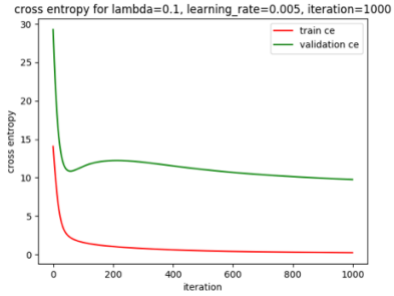


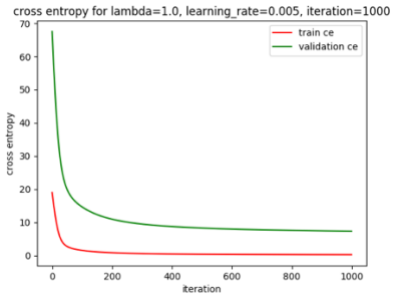
Result for Train_small data:











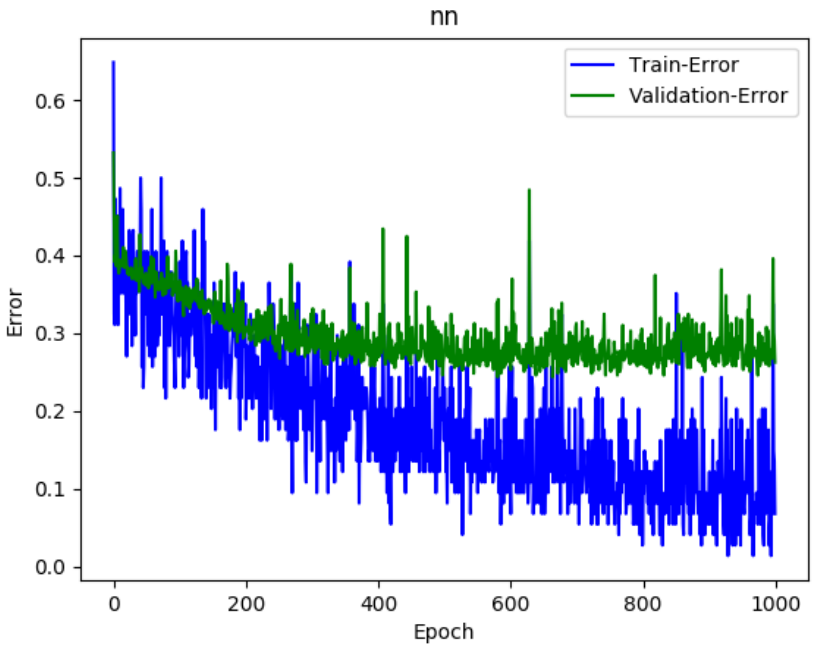
Comparing both the cross entropy and accuracy, we find that as we increase lambda the validation entropy decreases and accuracy increases, but the accuracy doesn't actually change much. Based on my experiments the **best penalty is: 1 for train_data, and 0.01 for train_small data**. A possible reason why model with penalty performs better is because there was an overfitting issue, and as we apply weight decay it actually penalize those weights that aren't key components.

The model with penalty performed a lot better on train_small compare to the model without penalty, this could be happening because small training images is more sensitive to overfitting problem, as the image are small, finding the key component is very important, and thus overfitting with confidence would cause big problem.

Code:

In [1]:
```
def logistic_pen(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.

    Outputs:
        f:              The sum of the loss over all data points. This is the objective that we want to minimize.
        df:             (M+1) x 1 vector of derivative of f w.r.t. weights.
    """
    N, M = data.shape[0], data.shape[1]
    wd = hyperparameters["weight_decay"]
    y = logistic_predict(weights, data)
    w_without_b = weights[:-1,:]
    penalty = (wd/2) * np.sum(w_without_b * w_without_b)
    constant = -((M/2) * np.log((2 * math.pi)/wd)) if wd != 0 else 0
    f = np.sum(-np.dot(targets.T,np.log(y))) - np.sum(np.dot((1-targets.T), np.log(1 - y))) + penalty + constant
    new_data = np.hstack((data, np.ones((N,1))))
    reg = np.pad(wd * w_without_b, ((0,1),(0,0)), 'constant')
    df = np.dot(new_data.T, (y - targets)) + reg
    return f, df, y
```

3. Neural Nets
    a. NN Output looks like this:



The train error is pretty high, train accuracy is very unstable, this might be caused by a learning that is too large.

Code:

```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.
    """
    ############################
    grad_relu = np.zeros(z.shape)
    grad_relu[np.where(z>0)]= 1
    grad_z = np.multiply(grad_h, grad_relu)
    return grad_z
    ############################
    # raise Exception('Not implemented')
```

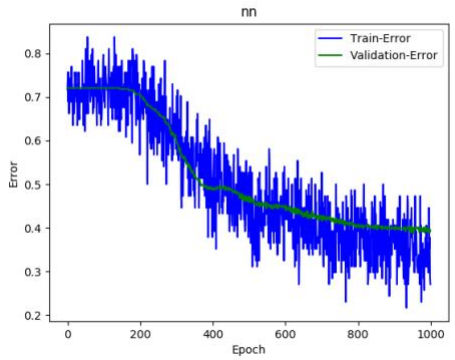
```python
def AffineBackward(grad_y, h, w):
    """Computes gradients of affine transformation.
    hint: you may need the matrix transpose np.dot(A,B).T = np.dot(B,A) and (A.T).T = A

    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_h = np.dot(grad_y, w.T)
    grad_w = np.dot(h.T, grad_y)
    grad_b = np.sum(grad_y, axis=0)
    return grad_h, grad_w, grad_b
    ############################
```

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

    Args:
        model:    Dictionary of all the weights.
        eps:      Learning rate.
        momentum: Momentum.
    """
    ############################
    model['v1'] = momentum * model['v1'] + (1-momentum) * model['dE_dW1']
    model['W1'] = model['W1'] - eps * model['v1']
    model['v2'] = momentum * model['v2'] + (1-momentum) * model['dE_dW2']
    model['W2'] = model['W2'] - eps * model['v2']
    model['v3'] = momentum * model['v3'] + (1-momentum) * model['dE_dW3']
     model['W3'] = model['W3'] - eps * model['v3']
    model['v1b'] = momentum * model['v1b'] + (1-momentum) * model['dE_db1']
    model['b1'] = model['b1'] - eps * model['v1b']
    model['v2b'] = momentum * model['v2b'] + (1-momentum) * model['dE_db2']
    model['b2'] = model['b2'] - eps * model['v2b']
    model['v3b'] = momentum * model['v3b'] + (1-momentum) * model['dE_db3']
    model['b3'] = model['b3'] - eps * model['v3b']
    ############################
    # raise Exception('Not implemented')
```

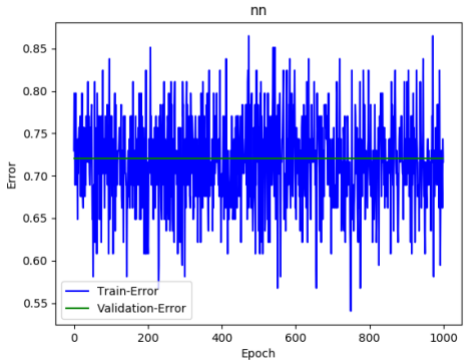
b. Since It's not required to show all result I'm just showing part of results that stands out.

1. Momentum , batch size fixed.

Esp = 0.001:



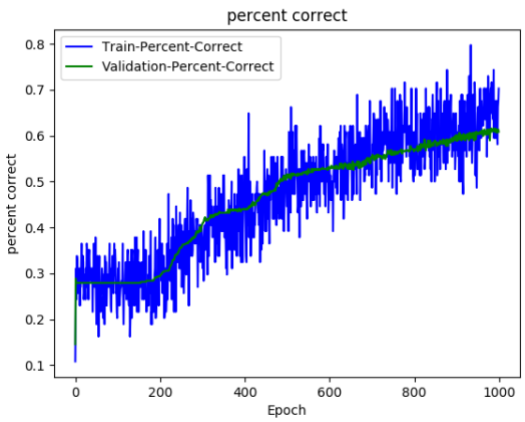
Esp=1.0:

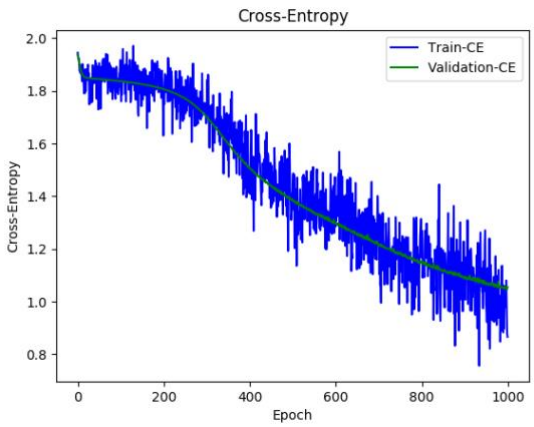


As we can see if we increase ESP from 0.001 to 1 the model's accuracy tends to converge worse, based on my experiment the best value for ESP is 0.001.

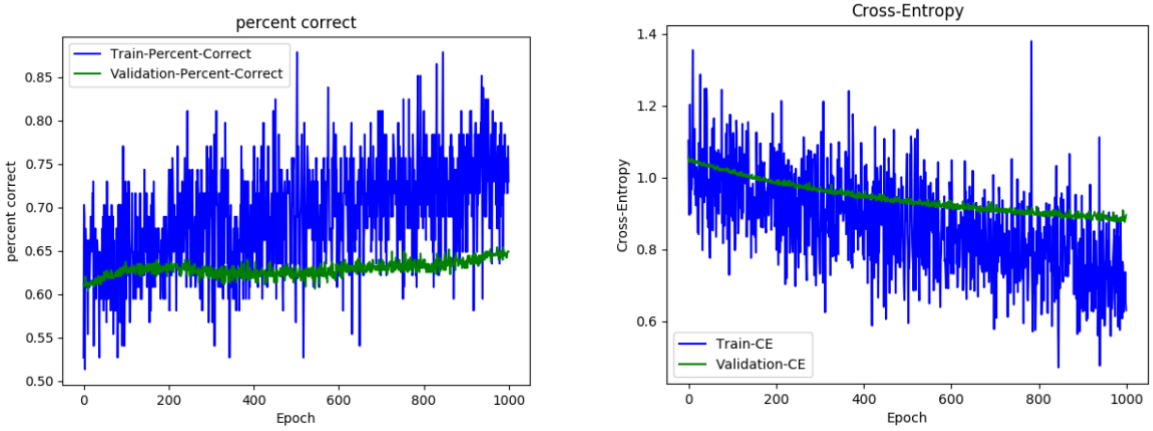
2. EPS fixed to 0.001, batch size fixed to 100

Momentum = 0:





Momentum = 0.45



As before I found when increasing the value of a momentum the model tends not to converge. So keep the momentum value low is probably a good idea.

Also after experiment with batch size I found that different bactch size can first affect the training spped, when set to 1 it will take a lot longer to train, and the result tends to be completely random. And when setting to 1000 there will be no change on neither training error nor validation error.

3. **By setting different hidden layers units (2, 60, 100), and reduced epoch to 200 for faster training, I observed that as we increase the number of hidden layer units the training error become a lot less and less noisy, but the validation error increases. Training error converges to a lower value and validation error converges to a higher value.**
   This could happen since more units means more likely to overfit, and thus validation performance will be worse than before, we want to generalize this model by penalizing hidden layer units.
4.