Homework 4

Zachary DeStefano, 15247592
CS 274B: Spring 2016

This is the code I used for prediction.

Code

```
num_iter = 10
num_points_per_iter = len(files)
stepSize = 0.01
lambdaVal = 0.01
totalPts = num_iter*num_points_per_iter

hammingLossValues = np.zeros(totalPts)
hammingLossAfterIter = np.zeros(num_iter)
hingeLossValues = np.zeros(totalPts)
hingeLossAfterIter = np.zeros(num_iter)

index = 0
perSymHingeLoss = 0
perSymHammingLoss = 0

for iter in range(num_iter):
    randomInds = np.random.permutation(len(files))
    randomIndsUse = randomInds[0:num_points_per_iter]
    for s in  randomIndsUse:
        # Load data ys,xs
        fh = open(datapath + files[s], 'r')
        rawlines = fh.readlines()
        lines = [line.strip('\n').split(',') for line in rawlines]
        fh.close()
        ys = [int(l[1]) - 1 for l in lines]
        xs = [[int(l[2]) - 1, int(l[3]), int(l[4]), int(l[5]) - 1, int(l[6]) - 1] for l in lines]

        ns = len(ys)

        #Define random variables for the inference process:
        Y = [gm.Var(i,10) for i in range(ns)]

                                #Construct the factors for prediction
        factors = []
        for ii in range(ns):
            curTable = np.matrix(np.zeros((10,1)))
            for ff in range(len(feature_sizes)):
                curTh = np.matrix(ThetaF[ff])
                curX = xs[ii][ff]
                curMat = np.matrix(curTh[:,curX])
                curTable = np.add(curTable,curMat)

            if ii<(ns-1):
                curMat = np.matrix(ThetaP[:,ys[ii+1]])
```

```
            curMat = np.reshape(curMat,(10,1))
            curTable = np.add(curTable,curMat)

        factors.append(gm.Factor([Y[ii]],curTable).exp())

    model_pred = gm.GraphModel(factors)

    # Copy factors and add extra Hamming factors for loss-augmented model
    factors_aug = [ f for f in factors ]
    factors_aug.extend( [gm.Factor([Y[i]], Loss[:,ys[i]]).exp() for i in range(ns)] )
    model_aug = gm.GraphModel(factors_aug)

    order = range(ns) # eliminate in sequence (Markov chain on y)
    wt = 1e-4 # for max elimination in JTree implementation

    # Now, the most likely configuration of the prediction model (for prediction) is:
    yhat_pred = gm.wmb.JTree(model_pred,order,wt).argmax()
                            yhatVals = yhat_pred.values()

    # and the maximizing argument of the loss (for computing the gradient) is
    yhat_aug = gm.wmb.JTree(model_aug,order,wt).argmax()
    yhatAugVals = yhat_aug.values()

    # use yhat_pred & ys to keep a running estimate of your prediction accuracy & print it
    #... # how often etc is up to you
    yhatVals = yhat_pred.values()
    hammingLoss = 0
    accuracy = 0
    for ii in range(ns):
        if(abs(yhatVals[ii]-ys[ii])>=1):
            hammingLoss += 1
        else:
            accuracy +=1
    print np.divide(np.double(accuracy),np.double(ns)) #print the accuracy

    #calculate hinge loss factors
    hingeLoss = 0
    hingeLoss += hammingLoss
    for ii in range(ns):
        for ff in range(len(feature_sizes)):
            curTh = np.matrix(ThetaF[ff])
            curX = xs[ii][ff]
            diffFactor = curTh[yhatAugVals[ii],curX]-curTh[ys[ii],curX]
            hingeLoss += diffFactor

        if ii < (ns - 1):
            otherDiff = ThetaP[yhatAugVals[ii], ys[ii + 1]]-ThetaP[ys[ii], ys[ii + 1]]
            hingeLoss += otherDiff
    for ii in range(10):
        for jj in range(10):
            hingeLoss += lambdaVal*(ThetaP[ii,jj]**2)
    for ff in range(numFeats):
        for ii in range(10):
            for jj in range(feature_sizes[ff]):
```

```
                    hingeLoss += lambdaVal*(ThetaF[ff][ii,jj]**2)

        # use yhat_aug & ys to update your parameters theta in the negative gradient direction
        for ii in range(ns-1):
            ThetaP[yhatAugVals[ii], yhatAugVals[ii + 1]] -= stepSize
            ThetaP[ys[ii],ys[ii+1]] += stepSize
        ThetaP = ThetaP - stepSize*lambdaVal * ThetaP

        for ii in range(ns):
            for ff in range(numFeats):
                ThetaF[ff][yhatAugVals[ii],xs[ii][ff]] -= stepSize
                ThetaF[ff][ys[ii], xs[ii][ff]] += stepSize
        for ff in range(numFeats):
            ThetaF[ff] = ThetaF[ff] - stepSize*lambdaVal * ThetaF[ff]

        perSymHingeLoss = np.divide(np.double(hingeLoss),np.double(ns))
        perSymHammingLoss = np.divide(np.double(hammingLoss), np.double(ns))
        hingeLossValues[index] = perSymHingeLoss
        hammingLossValues[index] = perSymHammingLoss
        index += 1
    hingeLossAfterIter[iter] = perSymHingeLoss
    hammingLossAfterIter[iter] = perSymHammingLoss
```

Results

After doing 10 passes through a random 100 points in each pass, these were the results:

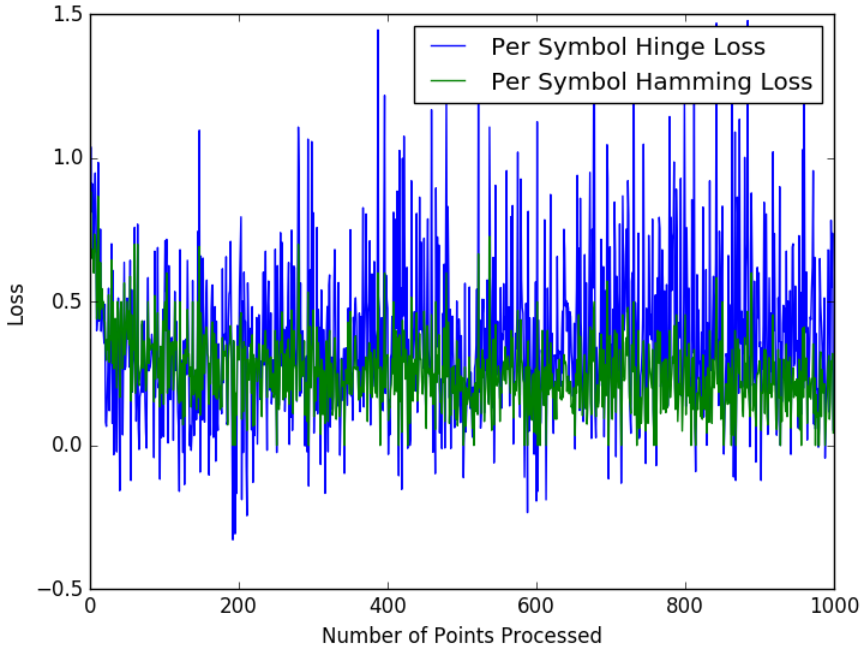

Figure 1: Hinge and Hamming Loss

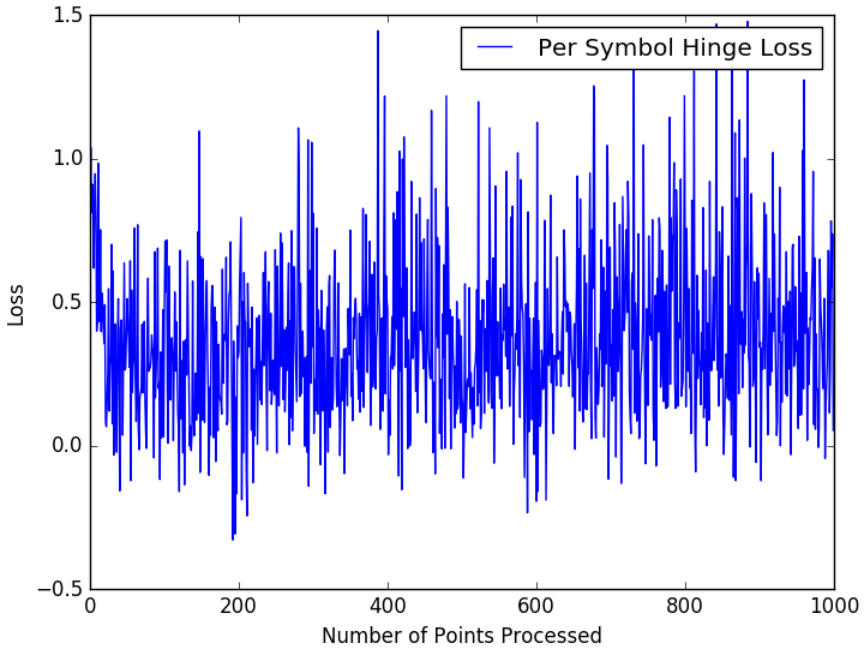

Figure 2: Hinge Loss

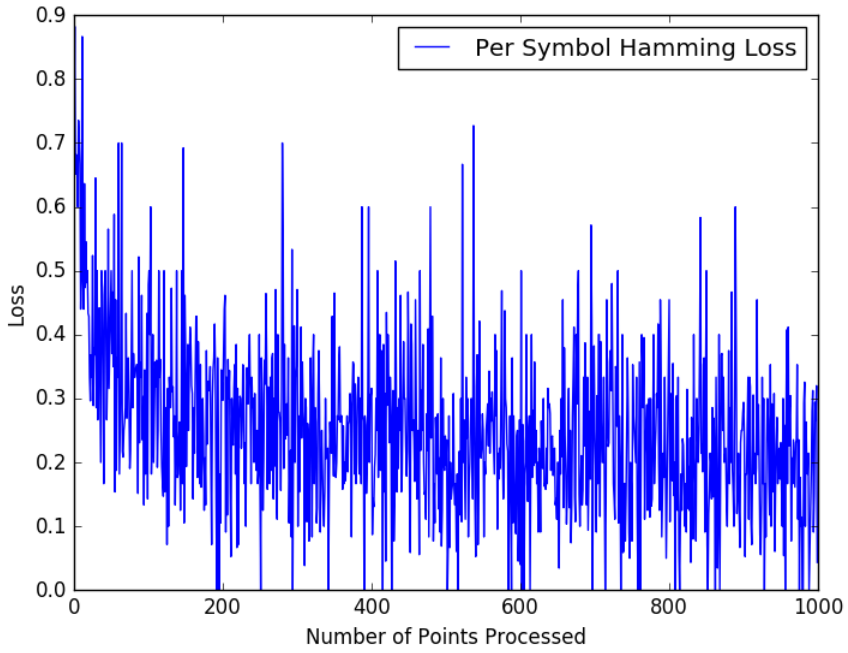

Figure 3: Hamming Loss