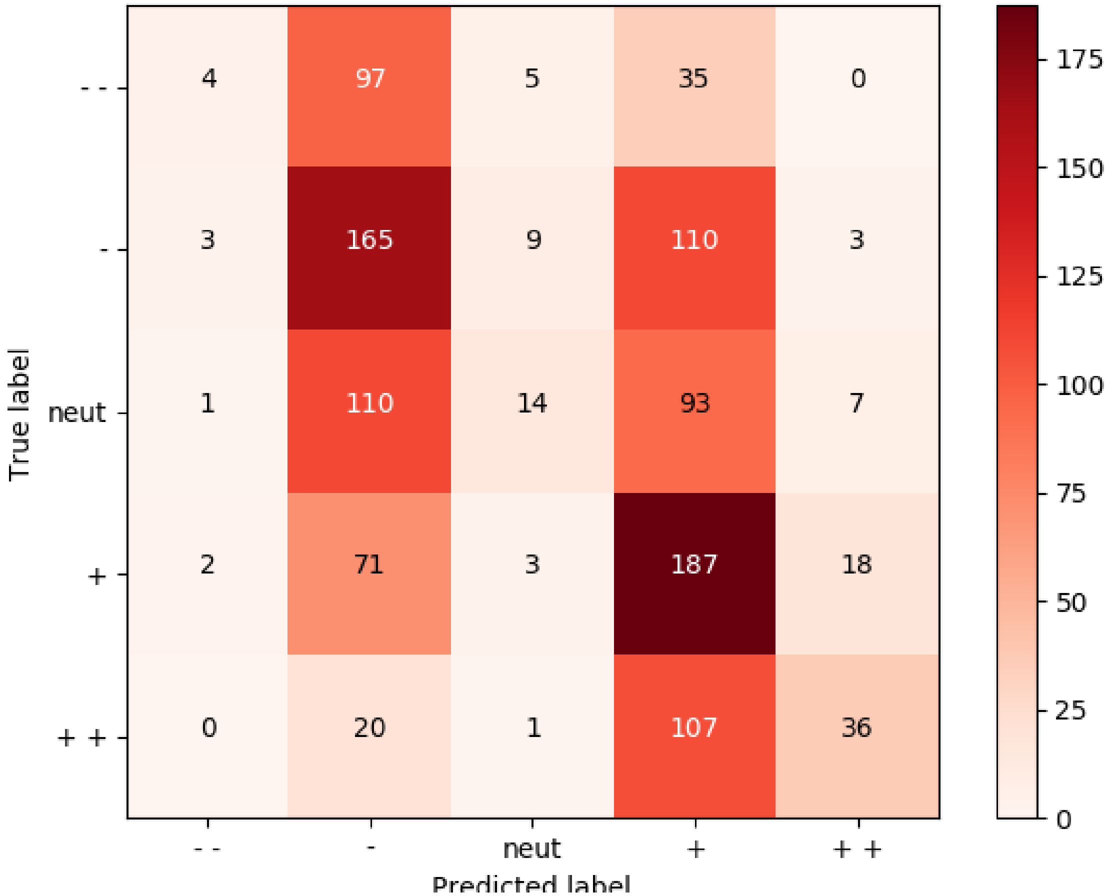
CS224n: Natural Language Processing with Deep Learning

Assignment 1 Coding Problems



```python
import numpy as np


def softmax(x):
    """Compute the softmax function for each row of the input x.

    It is crucial that this function is optimized for speed because it will be used
    frequently in later code. You might find numpy functions np.exp, np.sum, np.reshape,
    np.max, and numpy broadcasting useful for this task.

    You should also make sure that your code works for a single N-dimensional vector
    (treat the vector as a single row) and for M x N matrices.
    Also, make sure that the dimensions of the output match the input.

    Arguments:
    x -- A N dimensional vector or M x N dimensional numpy matrix.

    Return:
    x -- You are allowed to modify x in-place2
    """
    orig_shape = x.shape

    if len(x.shape) > 1:
        # Mat2rix
        tmp = np.max(x, axis = 1)
        x -= tmp.reshape((x.shape[0], 1))
        x = np.exp(x)
        tmp = np.sum(x, axis = 1)
        x /= tmp.reshape((x.shape[0], 1))

    else:
        # Vector
        tmp = np.max(x)
        x -= tmp
        x = np.exp(x)
        tmp = np.sum(x)
        x /= tmp

    assert x.shape == orig_shape
    return x
```

q1_softmax.py Efficiency Test

mean test time

```
def softmax(x):

    if len(x.shape) > 1:
        tmp = np.max(x, axis = 1)
        x -= tmp.reshape((x.shape[0], 1))
        x = np.exp(x)
        tmp = np.sum(x, axis = 1)
        x /= tmp.reshape((x.shape[0], 1))
    else:
        tmp = np.max(x)
        x -= tmp
        x = np.exp(x)
        tmp = np.sum(x)
        x /= tmp
    return x
```

0.408 sec

```
def softmax(x):

    assert len(x.shape) <= 2
    y = np.exp(x - np.max(x, axis=len(x.shape) - 1, keepdims=True))
    normalization = np.sum(y, axis=len(x.shape) - 1, keepdims=True)
    return np.divide(y, normalization)
```

0.401 sec

```
def softmax(x):

    log_c = np.max(x, axis=x.ndim - 1, keepdims=True)
    #for numerical stability
    y = np.sum(np.exp(x - log_c), axis=x.ndim - 1, keepdims=True)
    x = np.exp(x - log_c)/y
    return x
```

0.765 sec

```python
import numpy as np


def sigmoid(x):
    """
    Compute the sigmoid function for the input here.

    Arguments:
    x -- A scalar or numpy array.

    Return:
    s -- sigmoid(x)
    """

    ### YOUR CODE HERE
    s = 1. / (1 + np.exp(-x))
    ### END YOUR CODE

    return s


def sigmoid_grad(f):
    """
    Compute the gradient for the sigmoid function here. Note that
    for this implementation, the input s should be the sigmoid
    function value of your original input x.

    Arguments:
    s -- A scalar or numpy array.

    Return:
    ds -- Your computed gradient.
    """

    ### YOUR CODE HERE
    ds = f * (1-f)
    ### END YOUR CODE

    return ds
```

q2_gradcheck.py

```
import numpy as np
import random
from q2_sigmoid import sigmoid, sigmoid_grad

# First implement a gradient checker by filling in the following functions
def gradcheck_naive(f, x):
    """ Gradient check for a function f.

    Arguments:
    f -- a function that takes a single argument and outputs the
         cost and its gradients
    x -- the point (numpy array) to check the gradient at
    """

    rndstate = random.getstate()
    random.setstate(rndstate)
    fx, grad = f(x) # Evaluate function value at original point
    h = 1e-4        # Do not change this!

    # Iterate over all indexes in x
    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:
        ix = it.multi_index

        # Try modifying x[ix] with h defined above to compute
        # numerical gradients. Make sure you call random.setstate(rndstate)
        # before calling f(x) each time. This will make it possible
        # to test cost functions with built in randomness later.

        ### YOUR CODE HERE:
        x[ix] += h # increment by h
        random.setstate(rndstate)
        fxh, _ = f(x) # evalute f(x + h)
        x[ix] -= 2 * h # restore to previous value (very important!)
        random.setstate(rndstate)
        fxnh, _ = f(x)
        x[ix] += h
        numgrad = (fxh - fxnh) / 2 / h
```

```python
    ### END YOUR CODE

    # Compare gradients
    reldiff = abs(numgrad - grad[ix]) / max(1, abs(numgrad), abs(grad[ix]))
    if reldiff > 1e-5:
        print "Gradient check failed."
        print "First gradient error found at index %s" % str(ix)
        print "Your gradient: %f \t Numerical gradient: %f" % (
            grad[ix], numgrad)
        return

    it.iternext() # Step to next dimension

print "Gradient check passed!"
```

q2_neural.py

```
import numpy as np
import random
from q1_softmax import softmax
from q2_sigmoid import sigmoid, sigmoid_grad
from q2_gradcheck import gradcheck_naive


def forward_backward_prop(data, labels, params, dimensions):
    """
    Forward and backward propagation for a two-layer sigmoidal network

    Compute the forward propagation and for the cross entropy cost,
    and backward propagation for the gradients for all parameters.

    Arguments:
    data -- M x Dx matrix, where each row is a training example.
    labels -- M x Dy matrix, where each row is a one-hot vector.
    params -- Model parameters, these are unpacked for you.
    dimensions -- A tuple of input dimension, number of hidden units
                  and output dimension
    """

    ### Unpack network parameters (do not modify)
    ofs = 0
    Dx, H, Dy = (dimensions[0], dimensions[1], dimensions[2])

    W1 = np.reshape(params[ofs:ofs+ Dx * H], (Dx, H))
    ofs += Dx * H
    b1 = np.reshape(params[ofs:ofs + H], (1, H))
    ofs += H
    W2 = np.reshape(params[ofs:ofs + H * Dy], (H, Dy))
    ofs += H * Dy
    b2 = np.reshape(params[ofs:ofs + Dy], (1, Dy))

    ### YOUR CODE HERE: forward propagation
    hidden = sigmoid(data.dot(W1) + b1)
    prediction = softmax(hidden.dot(W2) + b2)
    cost = -np.sum(np.log(prediction) * labels)
    ### END YOUR CODE
```

```
### YOUR CODE HERE: backward propagation
delta = prediction - labels
gradW2 = hidden.T.dot(delta)
gradb2 = np.sum(delta, axis = 0)
delta = delta.dot(W2.T) * sigmoid_grad(hidden)
gradW1 = data.T.dot(delta)
gradb1 = np.sum(delta, axis = 0)
### END YOUR CODE


### Stack gradients (do not modify)
grad = np.concatenate((gradW1.flatten(), gradb1.flatten(),
    gradW2.flatten(), gradb2.flatten()))

return cost, grad
```

q3_word2vec.py

```python
import numpy as np
import random
from q1_softmax import softmax
from q2_gradcheck import gradcheck_naive
from q2_sigmoid import sigmoid, sigmoid_grad


def normalizeRows(x):
    """ Row normalization function

    Implement a function that normalizes each row of a matrix to have
    unit length.
    """

    ### YOUR CODE HERE
    N = x.shape[0]
    x /= np.sqrt(np.sum(x**2, axis=1)).reshape((N,1)) + 1e-30
    ### END YOUR CODE

    return x
```

```python
def softmaxCostAndGradient(predicted, target, outputVectors, dataset):

    """ Softmax cost function for word2vec models

    Implement the cost and gradients for one predicted word vector and one
    target word vector as a building block for word2vec models, assuming the
    softmax prediction function and cross entropy loss.

    Arguments:
    predicted -- numpy ndarray, predicted word vector (\hat{v})
    target -- integer, the index of the target word
    outputVectors -- "output" vectors (as rows) for all tokens
    dataset -- needed for negative sampling, unused here.

    Return:
    cost -- cross entropy cost for the softmax word prediction
    gradPred -- the gradient with respect to the predicted word vector
    grad -- the gradient with respect to all the other word vectors
    """

    ### YOUR CODE HERE
    probabilities = softmax(predicted.dot(outputVectors.T))
    cost = -np.log(probabilities[target])
    delta = probabilities
    delta[target] -= 1
    N = delta.shape[0]
    D = predicted.shape[0]
    grad = delta.reshape((N,1)) * predicted.reshape((1,D))
    gradPred = (delta.reshape((1,N)).dot(outputVectors)).flatten()
    ### END YOUR CODE

    return cost, gradPred, grad
```

```
def getNegativeSamples(target, dataset, K):

    """ Samples K indexes which are not the target """

    indices = [None] * K
    for k in xrange(K):
        newidx = dataset.sampleTokenIdx()
        while newidx == target:
            newidx = dataset.sampleTokenIdx()
        indices[k] = newidx
    return indices
```

```
def negSamplingCostAndGradient(predicted, target, outputVectors, dataset, K=10):

    """ Negative sampling cost function for word2vec models
    Implement the cost and gradients for one predicted word vector
    and one target word vector as a building block for word2vec
    models, using the negative sampling technique. K is the sample size.
    Note: See test_word2vec below for dataset's initialization.

    Arguments/Return Specifications: same as softmaxCostAndGradient
    """

    # Sampling of indices is done for you. Do not modify this!
    indices = [target]
    indices.extend(getNegativeSamples(target, dataset, K))

    ### YOUR CODE HERE
    grad = np.zeros(outputVectors.shape)
    gradPred = np.zeros(predicted.shape)

    indices = [target]
    for k in xrange(K):
        newidx = dataset.sampleTokenIdx()
        while newidx == target:
            newidx = dataset.sampleTokenIdx()
        indices += [newidx]

    labels = np.array([1] + [-1 for k in xrange(K)])
    vecs = outputVectors[indices,:]

    t = sigmoid(vecs.dot(predicted) * labels)
    cost = -np.sum(np.log(t))

    delta = labels * (t - 1)
    gradPred = delta.reshape((1,K+1)).dot(vecs).flatten()
    gradtemp = delta.reshape((K+1,1)).dot(predicted.reshape(
        (1,predicted.shape[0])))
    for k in xrange(K+1):
        grad[indices[k]] += gradtemp[k,:]
    ### END YOUR CODE

    return cost, gradPred, grad
```

```
def skipgram(currentWord, C, contextWords, tokens, inputVectors, outputVectors,
             dataset, word2vecCostAndGradient=softmaxCostAndGradient):
    """ Skip-gram model in word2vec

    Implement the skip-gram model in this function.

    Arguments:
    currrentWord -- a string of the current center word
    C -- integer, context size
    contextWords -- list of no more than 2*C strings, the context words
    tokens -- a dictionary that maps words to their indices in the word vector list
    inputVectors -- "input" word vectors (as rows) for all tokens
    outputVectors -- "output" word vectors (as rows) for all tokens
    word2vecCostAndGradient -- the cost and gradient function for a prediction vector given the target
                               word vectors, could be one of the two cost functions you implemented above.

    Return:
    cost -- the cost function value for the skip-gram model
    grad -- the gradient with respect to the word vectors
    """

    cost = 0.0
    gradIn = np.zeros(inputVectors.shape)
    gradOut = np.zeros(outputVectors.shape)

    currentI = tokens[currentWord]
    predicted = inputVectors[currentI, :]

    cost = 0.0
    gradIn = np.zeros(inputVectors.shape)
    gradOut = np.zeros(outputVectors.shape)
    for cwd in contextWords:
        idx = tokens[cwd]
        cc, gp, gg = word2vecCostAndGradient(predicted, idx, outputVectors, dataset)
        cost += cc
        gradOut += gg
        gradIn[currentI, :] += gp

    return cost, gradIn, gradOut
```

```python
def cbow(currentWord, C, contextWords, tokens, inputVectors, outputVectors,
         dataset, word2vecCostAndGradient=softmaxCostAndGradient):
    """CBOW model in word2vec

    Implement the continuous bag-of-words model in this function.

    Arguments/Return specifications: same as the skip-gram model

    Extra credit: Implementing CBOW is optional, but the gradient
    derivations are not. If you decide not to implement CBOW, remove
    the NotImplementedError.
    """

    cost = 0.0
    gradIn = np.zeros(inputVectors.shape)
    gradOut = np.zeros(outputVectors.shape)

    ### YOUR CODE HERE
    D = inputVectors.shape[1]
    predicted = np.zeros((D,))

    indices = [tokens[cwd] for cwd in contextWords]
    for idx in indices:
        predicted += inputVectors[idx, :]

    cost, gp, gradOut = word2vecCostAndGradient(predicted, tokens[currentWord], outputVectors, dataset)
    gradIn = np.zeros(inputVectors.shape)
    for idx in indices:
        gradIn[idx, :] += gp
    ### END YOUR CODE

    return cost, gradIn, gradOut
```

```
#############################################
# Testing functions below. DO NOT MODIFY!   #
#############################################

def word2vec_sgd_wrapper(word2vecModel, tokens, wordVectors, dataset, C,
                         word2vecCostAndGradient=softmaxCostAndGradient):
    batchsize = 50
    cost = 0.0
    grad = np.zeros(wordVectors.shape)
    N = wordVectors.shape[0]
    inputVectors = wordVectors[:N/2,:]
    outputVectors = wordVectors[N/2:,:]
    for i in xrange(batchsize):
        C1 = random.randint(1,C)
        centerword, context = dataset.getRandomContext(C1)

        if word2vecModel == skipgram:
            denom = 1
        else:
            denom = 1

        c, gin, gout = word2vecModel(
            centerword, C1, context, tokens, inputVectors, outputVectors,
            dataset, word2vecCostAndGradient)
        cost += c / batchsize / denom
        grad[:N/2, :] += gin / batchsize / denom
        grad[N/2:, :] += gout / batchsize / denom

    return cost, grad
```

q3_sgd.py

```
# Save parameters every a few SGD iterations as fail-safe

SAVE_PARAMS_EVERY = 5000

import glob
import random
import numpy as np
import os.path as op
import cPickle as pickle


def load_saved_params():
    """
    A helper function that loads previously saved parameters and resets iteration start.
    """
    st = 0
    for f in glob.glob("saved_params_*.npy"):
        iter = int(op.splitext(op.basename(f))[0].split("_")[2])
        if (iter > st):
            st = iter

    if st > 0:
        with open("saved_params_%d.npy" % st, "r") as f:
            params = pickle.load(f)
            state = pickle.load(f)
        return st, params, state
    else:
        return st, None, None


def save_params(iter, params):
    with open("saved_params_%d.npy" % iter, "w") as f:
        pickle.dump(params, f)
        pickle.dump(random.getstate(), f)
```

```python
def sgd(f, x0, step, iterations, postprocessing=None, useSaved=False,
        PRINT_EVERY=10):
    """ Stochastic Gradient Descent

    Implement the stochastic gradient descent method in this function.

    Arguments:
    f -- the function to optimize, it should take a single argument and yield two outputs,
         a cost and the gradient with respect to the arguments
    x0 -- the initial point to start SGD from
    step -- the step size for SGD
    iterations -- total iterations to run SGD for
    postprocessing -- postprocessing function for the parameters if necessary.
                      In the case of word2vec we will need to
                      normalize the word vectors to have unit length.
    PRINT_EVERY -- specifies how many iterations to output loss

    Return:
    x -- the parameter value after SGD finishes
    """

    # Anneal learning rate every several iterations
    ANNEAL_EVERY = 20000

    if useSaved:
        start_iter, oldx, state = load_saved_params()
        if start_iter > 0:
            x0 = oldx
            step *= 0.5 ** (start_iter / ANNEAL_EVERY)

        if state:
            random.setstate(state)
    else:
        start_iter = 0

    x = x0

    if not postprocessing:
        postprocessing = lambda x: x
```

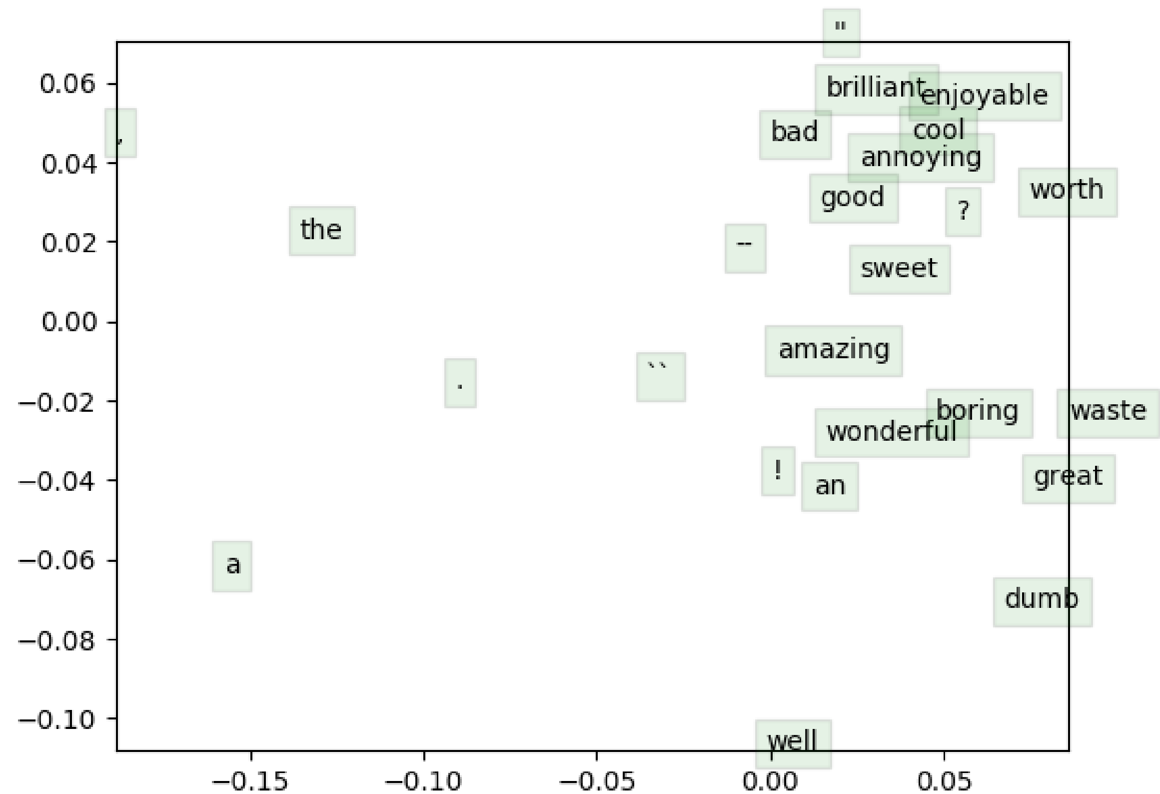
```
expcost = None

for iter in xrange(start_iter + 1, iterations + 1):
    # Don't forget to apply the postprocessing after every iteration!
    # You might want to print the progress every few iterations.

    cost = None

    ### YOUR CODE HERE
    cost, grad = f(x)
    x -= step * grad

    x = postprocessing(x)

    ### END YOUR CODE

    if iter % PRINT_EVERY == 0:
        if not expcost:
            expcost = cost
        else:
            expcost = .95 * expcost + .05 * cost
        print "iter %d: %f" % (iter, expcost)

    if iter % SAVE_PARAMS_EVERY == 0 and useSaved:
        save_params(iter, x)

    if iter % ANNEAL_EVERY == 0:
        step *= 0.5

return x
```

q3_run.py

```
import random
import numpy as np
from utils.treebank import StanfordSentiment
import matplotlib
matplotlib.use('agg')
import matplotlib.pyplot as plt
import time
from q3_word2vec import *
from q3_sgd import *

# Reset the random seed to make sure that everyone gets the same results
random.seed(314)
dataset = StanfordSentiment()
tokens = dataset.tokens()
nWords = len(tokens)

# We are going to train 10-dimensional vectors for this assignment
dimVectors = 10

# Context size
C = 5

# Reset the random seed to make sure that everyone gets the same results
random.seed(31415)
np.random.seed(9265)

startTime=time.time()
wordVectors = np.concatenate(
    ((np.random.rand(nWords, dimVectors) - 0.5) /
       dimVectors, np.zeros((nWords, dimVectors))), axis=0)

wordVectors = sgd(lambda vec: word2vec_sgd_wrapper(skipgram, tokens, vec, dataset, C,
        negSamplingCostAndGradient),
    wordVectors, 0.3, 40000, None, True, PRINT_EVERY=10)

# NOTE: You must physically replace 'skipgram' with 'cbow' to run that model.
# Note that normalization is not called here. This is not a bug,
# normalizing during training loses the notion of length.
```

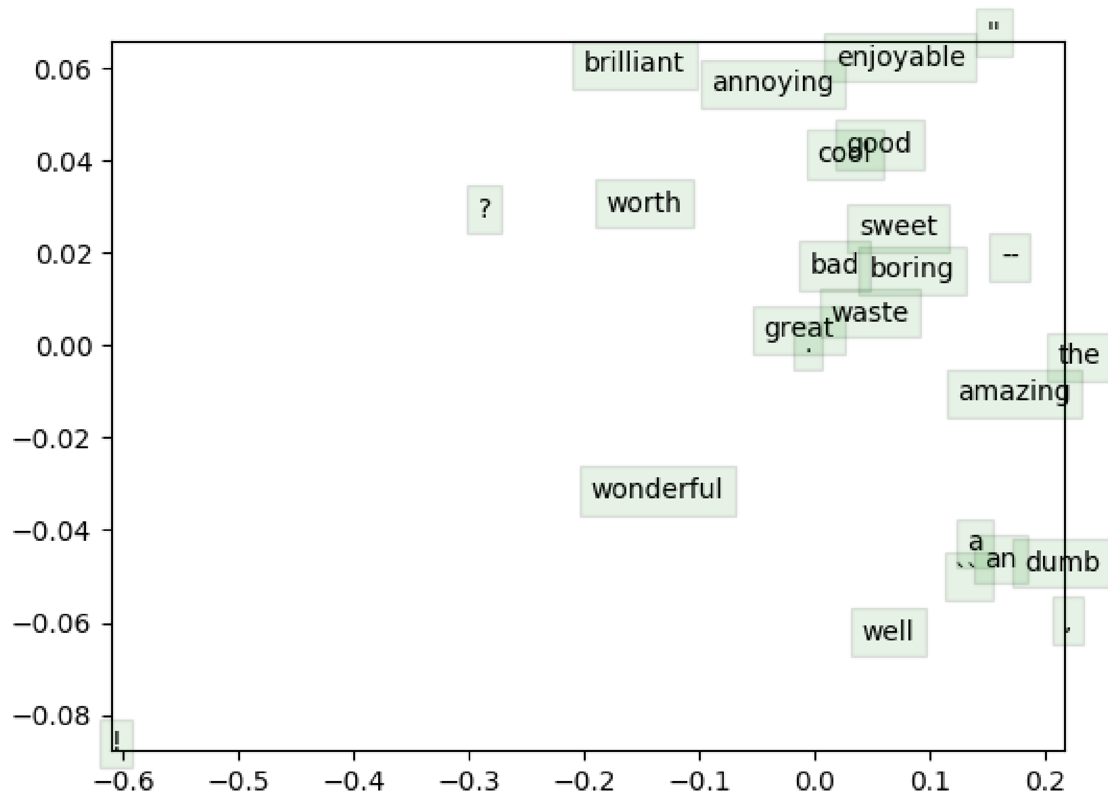
```
print "sanity check: cost at convergence should be around or below 10"
print "training took %d seconds" % (time.time() - startTime)

# concatenate the input and output word vectors
wordVectors = np.concatenate(
    (wordVectors[:nWords,:], wordVectors[nWords:,:]),
    axis=0)
# wordVectors = wordVectors[:nWords,:] + wordVectors[nWords:,:]

visualizeWords = [
    "the", "a", "an", ",", ".", "?", "!", "``", "''", "--",
    "good", "great", "cool", "brilliant", "wonderful", "well", "amazing",
    "worth", "sweet", "enjoyable", "boring", "bad", "waste", "dumb",
    "annoying"]

visualizeIdx = [tokens[word] for word in visualizeWords]
visualizeVecs = wordVectors[visualizeIdx, :]
temp = (visualizeVecs - np.mean(visualizeVecs, axis=0))
covariance = 1.0 / len(visualizeIdx) * temp.T.dot(temp)
U,S,V = np.linalg.svd(covariance)
coord = temp.dot(U[:,0:2])

for i in xrange(len(visualizeWords)):
    plt.text(coord[i,0], coord[i,1], visualizeWords[i],
        bbox=dict(facecolor='green', alpha=0.1))

plt.xlim((np.min(coord[:,0]), np.max(coord[:,0])))
plt.ylim((np.min(coord[:,1]), np.max(coord[:,1])))

plt.savefig('q3_word_vectors.png')
```

2-D visualization of skip-gram model output



Cost function appeared to converge at ≈ 9.4 (40k iterations)

2-D visualization of cbow model output



q4_sentiment.py

```
import argparse
import numpy as np
import matplotlib
matplotlib.use('agg')
import matplotlib.pyplot as plt
import itertools

from utils.treebank import StanfordSentiment
import utils.glove as glove

from q3_sgd import load_saved_params, sgd

# We will use sklearn here because it will run faster than implementing
# ourselves. However, for other parts of this assignment you must implement
# the functions yourself!
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix


def getArguments():
    parser = argparse.ArgumentParser()
    group = parser.add_mutually_exclusive_group(required=True)
    group.add_argument("--pretrained", dest="pretrained", action="store_true",
                       help="Use pretrained GloVe vectors.")
    group.add_argument("--yourvectors", dest="yourvectors", action="store_true",
                       help="Use your vectors from q3.")
    return parser.parse_args()
```

```
def getSentenceFeatures(tokens, wordVectors, sentence):
    """
    Obtain the sentence feature for sentiment analysis by averaging its
    word vectors
    """

    # Implement computation for the sentence features given a sentence.

    # Inputs:
    # tokens -- a dictionary that maps words to their indices in the word vector list
    # wordVectors -- word vectors (each row) for all tokens
    # sentence -- a list of words in the sentence of interest

    # Output:
    # - sentVector: feature vector for the sentence

    sentVector = np.zeros((wordVectors.shape[1],))

    ### YOUR CODE HERE
    indices = [tokens[word] for word in sentence]
    sentVector = np.mean(wordVectors[indices, :], axis=0)
    ### END YOUR CODE

    assert sentVector.shape == (wordVectors.shape[1],)
    return sentVector



def getRegularizationValues():

    """Try different regularizations
    Return a sorted list of values to try.
    """
    values = None   # Assign a list of floats in the block below
    ### YOUR CODE HERE
    values = np.logspace(-6,0.1,21)
    ### END YOUR CODE
    return sorted(values)
```

```python
def chooseBestModel(results):
    """Choose the best model based on parameter tuning on the dev set

    Arguments:
    results -- A list of python dictionaries of the following format:
        {
            "reg": regularization,
            "clf": classifier,
            "train": trainAccuracy,
            "dev": devAccuracy,
            "test": testAccuracy
        }

    Returns:
    Your chosen result dictionary.
    """
    bestResult = []
    ### YOUR CODE HERE
    sorted_results = sorted(results, key=lambda x: x['dev'],reverse=True)
    bestResult = sorted_results[0]
    ### END YOUR CODE

    return bestResult


def accuracy(y, yhat):
    """ Precision for classifier """
    assert(y.shape == yhat.shape)
    return np.sum(y == yhat) * 100.0 / y.size


def plotRegVsAccuracy(regValues, results, filename):
    """ Make a plot of regularization vs accuracy """
    plt.plot(regValues, [x["train"] for x in results])
    plt.plot(regValues, [x["dev"] for x in results])
    plt.xscale('log')
    plt.xlabel("regularization")
    plt.ylabel("accuracy")
    plt.legend(['train', 'dev'], loc='upper left')
    plt.savefig(filename)
```

```python
def outputConfusionMatrix(features, labels, clf, filename):
    """ Generate a confusion matrix """
    pred = clf.predict(features)
    cm = confusion_matrix(labels, pred, labels=range(5))
    plt.figure()
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Reds)
    plt.colorbar()
    classes = ["- -", "-", "neut", "+", "+ +"]
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes)
    plt.yticks(tick_marks, classes)
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.savefig(filename)


def outputPredictions(dataset, features, labels, clf, filename):
    """ Write the predictions to file """
    pred = clf.predict(features)
    with open(filename, "w") as f:
        print >> f, "True\tPredicted\tText"
        for i in xrange(len(dataset)):
            print >> f, "%d\t%d\t%s" % (
                labels[i], pred[i], " ".join(dataset[i][0]))
```

```python
def main(args):
    """ Train a model to do sentiment analyis"""

    # Load the dataset
    dataset = StanfordSentiment()
    tokens = dataset.tokens()
    nWords = len(tokens)

    if args.yourvectors:
        _, wordVectors, _ = load_saved_params()
        wordVectors = np.concatenate(
            (wordVectors[:nWords,:], wordVectors[nWords:,:]),
            axis=1)
    elif args.pretrained:
        wordVectors = glove.loadWordVectors(tokens)
    dimVectors = wordVectors.shape[1]

    # Load the train set
    trainset = dataset.getTrainSentences()
    nTrain = len(trainset)
    trainFeatures = np.zeros((nTrain, dimVectors))
    trainLabels = np.zeros((nTrain,), dtype=np.int32)
    for i in xrange(nTrain):
        words, trainLabels[i] = trainset[i]
        trainFeatures[i, :] = getSentenceFeatures(tokens, wordVectors, words)

    # Prepare dev set features
    devset = dataset.getDevSentences()
    nDev = len(devset)
    devFeatures = np.zeros((nDev, dimVectors))
    devLabels = np.zeros((nDev,), dtype=np.int32)
    for i in xrange(nDev):
        words, devLabels[i] = devset[i]
        devFeatures[i, :] = getSentenceFeatures(tokens, wordVectors, words)
```

```
# Prepare test set features
testset = dataset.getTestSentences()
nTest = len(testset)
testFeatures = np.zeros((nTest, dimVectors))
testLabels = np.zeros((nTest,), dtype=np.int32)
for i in xrange(nTest):
    words, testLabels[i] = testset[i]
    testFeatures[i, :] = getSentenceFeatures(tokens, wordVectors, words)

# We will save our results from each run
results = []
regValues = getRegularizationValues()
for reg in regValues:
    print "Training for reg=%f" % reg
    # Note: add a very small number to regularization to please the library
    clf = LogisticRegression(C=1.0/(reg + 1e-12))
    clf.fit(trainFeatures, trainLabels)

    # Test on train set
    pred = clf.predict(trainFeatures)
    trainAccuracy = accuracy(trainLabels, pred)
    print "Train accuracy (%%): %f" % trainAccuracy

    # Test on dev set
    pred = clf.predict(devFeatures)
    devAccuracy = accuracy(devLabels, pred)
    print "Dev accuracy (%%): %f" % devAccuracy

    # Test on test set
    # Note: always running on test is poor style. Typically, you should
    # do this only after validation.
    pred = clf.predict(testFeatures)
    testAccuracy = accuracy(testLabels, pred)
    print "Test accuracy (%%): %f" % testAccuracy

    results.append({
        "reg": reg,
        "clf": clf,
        "train": trainAccuracy,
        "dev": devAccuracy,
        "test": testAccuracy})
```

```
    # Print the accuracies
    print ""
    print "=== Recap ==="
    print "Reg\t\tTrain\tDev\tTest"
    for result in results:
        print "%.2E\t%.3f\t%.3f\t%.3f" % (
            result["reg"],
            result["train"],
            result["dev"],
            result["test"])
    print ""



    bestResult = chooseBestModel(results)
    print "Best regularization value: %0.2E" % bestResult["reg"]
    print "Test accuracy (%%): %f" % bestResult["test"]

    # do some error analysis
    if args.pretrained:
        plotRegVsAccuracy(regValues, results, "q4_reg_v_acc.png")
        outputConfusionMatrix(devFeatures, devLabels, bestResult["clf"],
                              "q4_dev_conf.png")
        outputPredictions(devset, devFeatures, devLabels, bestResult["clf"],
                          "q4_dev_pred.txt")


if __name__ == "__main__":
    main(getArguments())
```

klh@INS:~/Documents/assignment1$ **python2.7 q4_sentiment.py --yourvectors**

Reg	Train	Dev	Test
0.00E+00	30.665	30.609	29.593
1.00E-06	30.630	30.699	29.548
2.02E-06	30.641	30.699	29.548
4.07E-06	30.665	30.609	29.593
8.22E-06	30.676	30.609	29.593
1.66E-05	30.665	30.609	29.548
3.35E-05	30.665	30.609	29.593
6.76E-05	30.641	30.699	29.593
1.36E-04	30.712	30.609	29.638
2.75E-04	30.700	30.518	29.502
5.56E-04	30.712	30.790	29.412
1.12E-03	30.770	30.699	29.367
2.26E-03	30.735	30.609	29.412
4.57E-03	30.583	30.790	29.140
9.23E-03	30.478	30.972	29.457
1.86E-02	30.559	30.790	29.095
3.76E-02	30.559	30.245	29.095
7.59E-02	30.150	30.790	28.824
1.53E-01	29.951	30.790	28.688
3.09E-01	29.412	30.881	27.873
6.24E-01	29.026	29.428	26.244
1.26E+00	28.207	27.066	25.158

Best regularization value: 9.23E-03
Test accuracy (%): 29.457014

Accuracy not much influenced by regularization over a broad range of values …

```
klh@INS:~/Documents/assignment1$ python2.7 q4_sentiment.py --yourvectors
```

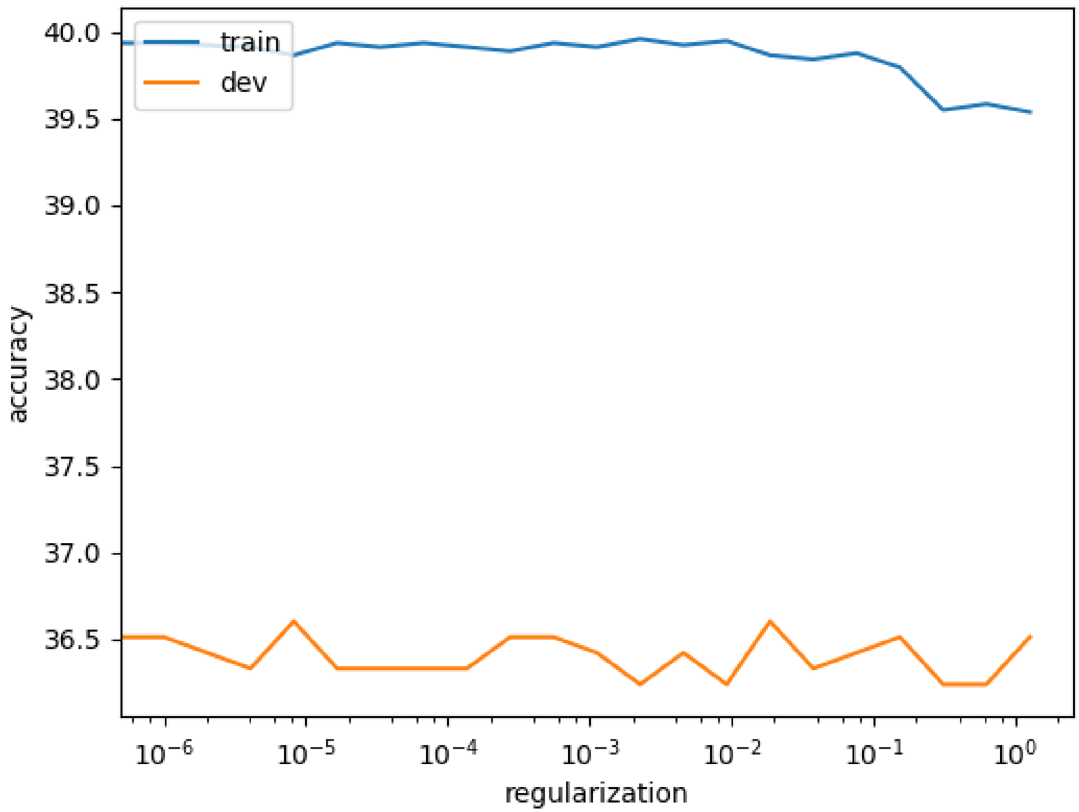
Reg	Train	Dev	Test
1.00E-06	30.630	30.699	29.548
2.15E-06	30.676	30.609	29.593
4.64E-06	30.653	30.699	29.548
1.00E-05	30.700	30.609	29.548
2.15E-05	30.665	30.609	29.593
4.64E-05	30.641	30.699	29.548
1.00E-04	30.676	30.609	29.593
2.15E-04	30.641	30.609	29.593
4.64E-04	30.700	30.699	29.502
1.00E-03	30.758	30.609	29.593
2.15E-03	30.723	30.699	29.502
4.64E-03	30.583	30.699	29.140
1.00E-02	30.478	30.972	29.502
2.15E-02	30.548	30.518	28.914
4.64E-02	30.454	30.064	29.095
1.00E-01	29.951	30.609	28.778
2.15E-01	29.822	30.790	28.462
4.64E-01	29.319	29.973	27.195
1.00E+00	28.453	27.975	25.158
2.15E+00	27.762	26.340	24.434
4.64E+00	27.294	25.431	23.167
1.00E+01	27.235	25.522	23.077
2.15E+01	27.235	25.522	23.032
4.64E+01	27.247	25.522	23.032
1.00E+02	27.247	25.522	23.032

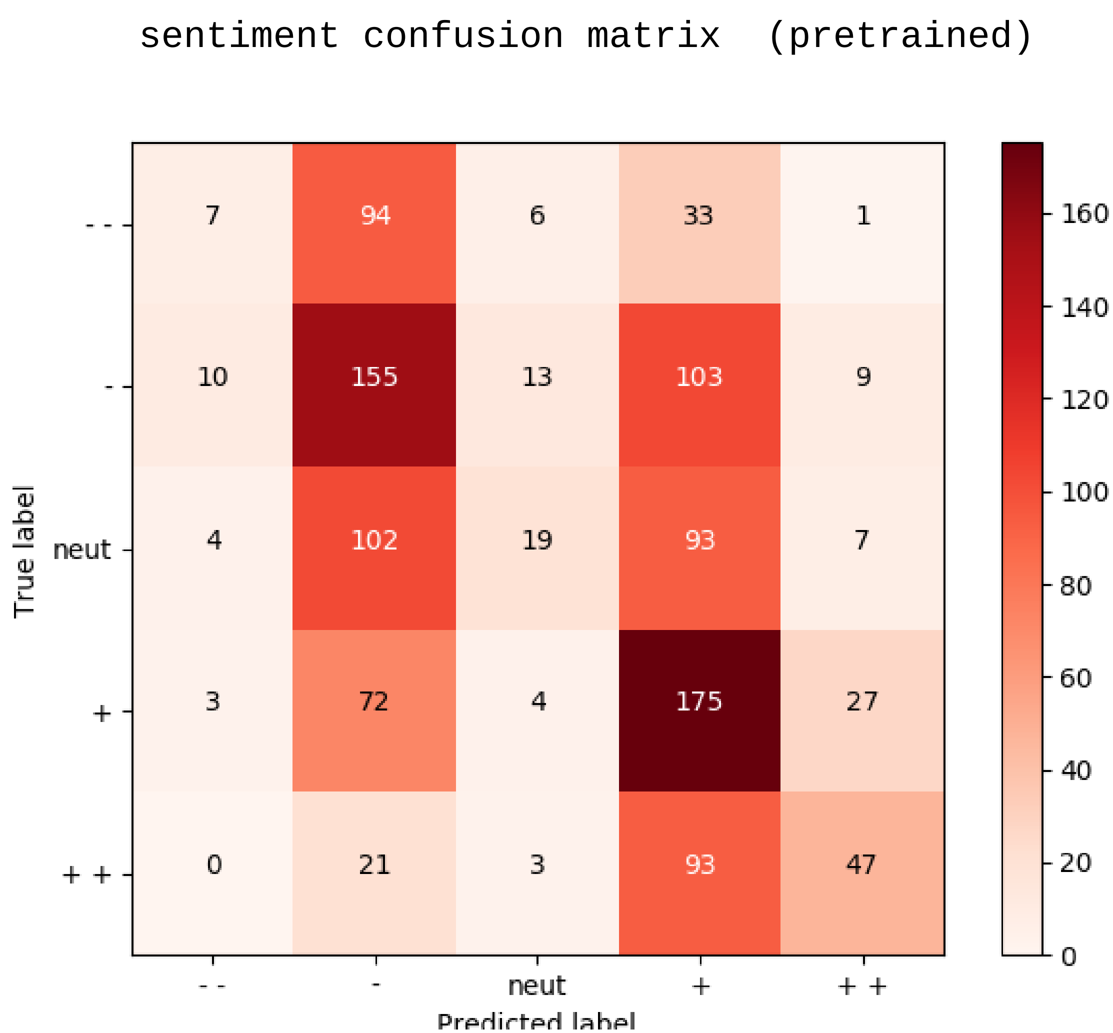
```
Best regularization value: 1.00E-02
Test accuracy (%): 29.502262
```

```
klh@INS:~/Documents/assignment1$ python2.7 q4_sentiment.py --pretrained

=== Recap ===
Reg         Train       Dev         Test
0.00E+00    39.934      36.331      36.968
1.00E-06    39.934      36.512      37.014
2.02E-06    39.923      36.421      36.968
4.07E-06    39.911      36.331      37.014
8.22E-06    39.864      36.603      37.104
1.66E-05    39.934      36.331      36.878
3.35E-05    39.911      36.331      36.923
6.76E-05    39.934      36.331      36.878
1.36E-04    39.911      36.331      37.014
2.75E-04    39.888      36.512      37.059
5.56E-04    39.934      36.512      37.014
1.12E-03    39.911      36.421      37.059
2.26E-03    39.958      36.240      36.968
4.57E-03    39.923      36.421      37.059
9.23E-03    39.946      36.240      37.195
1.86E-02    39.864      36.603      37.240
3.76E-02    39.841      36.331      37.511
7.59E-02    39.876      36.421      37.195
1.53E-01    39.794      36.512      37.466
3.09E-01    39.548      36.240      37.285
6.24E-01    39.583      36.240      37.240
1.26E+00    39.537      36.512      37.330

Best regularization value: 8.22E-06
Test accuracy (%): 37.104072
```

Accuracy not much influenced by regularization over a broad range of values …

Regularization vs. Accuracy (pre-trained)




sentiment confusion matrix (pretrained)
True label
- - -
- -
neut
+
+ +
7
94
6
33
1
10
155
13
103
9
4
102
19
93
7
3
72
4
175
27
0
21
3
93
47
- -
-
neut
+
+ +
Predicted label
160
140
120
100
80
60
40
20
0

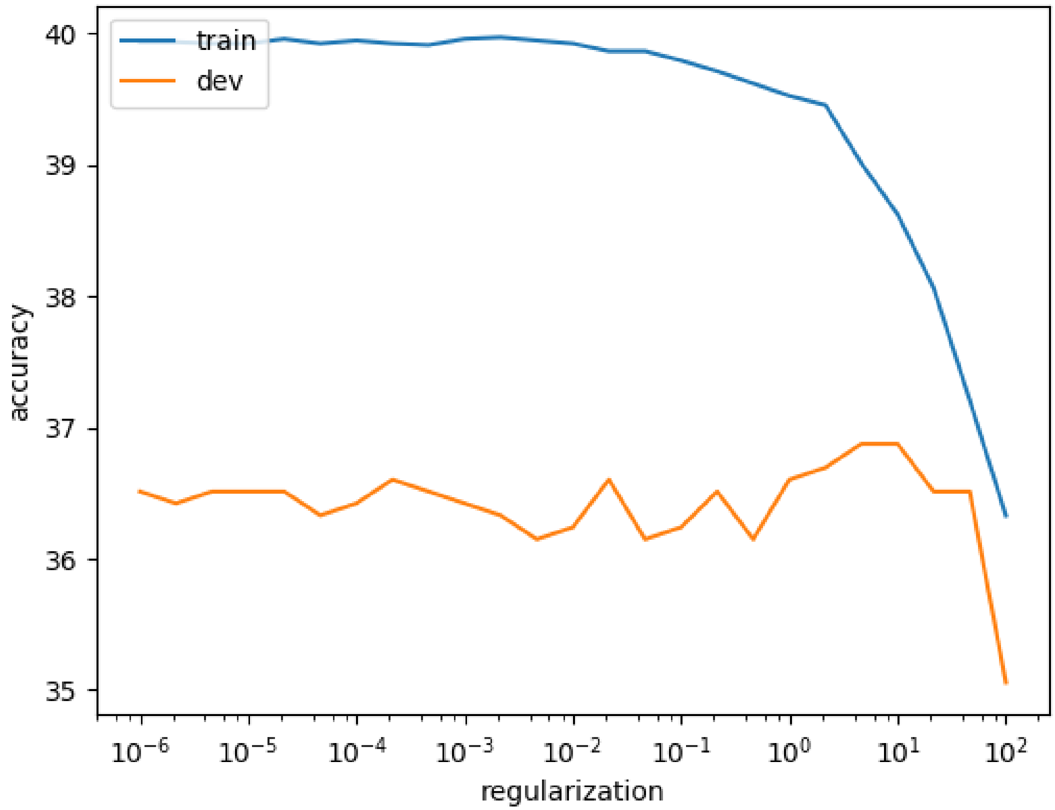
```
klh@INS:~/Documents/assignment1$ python2.7 q4_sentiment.py --pretrained
```

Reg	Train	Dev	Test
1.00E-06	39.934	36.512	37.014
2.15E-06	39.934	36.421	37.104
4.64E-06	39.923	36.512	37.104
1.00E-05	39.923	36.512	37.014
2.15E-05	39.958	36.512	37.014
4.64E-05	39.923	36.331	36.923
1.00E-04	39.946	36.421	36.968
2.15E-04	39.923	36.603	37.014
4.64E-04	39.911	36.512	37.059
1.00E-03	39.958	36.421	36.968
2.15E-03	39.970	36.331	36.968
4.64E-03	39.946	36.149	37.059
1.00E-02	39.923	36.240	37.195
2.15E-02	39.864	36.603	37.240
4.64E-02	39.864	36.149	37.466
1.00E-01	39.794	36.240	37.149
2.15E-01	39.712	36.512	37.240
4.64E-01	39.618	36.149	37.285
1.00E+00	39.525	36.603	37.330
2.15E+00	39.455	36.694	37.285
4.64E+00	39.010	36.876	37.285
1.00E+01	38.624	36.876	37.692
2.15E+01	38.062	36.512	37.014
4.64E+01	37.207	36.512	36.154
1.00E+02	36.330	35.059	35.701

Best regularization value: 4.64E+00

Test accuracy (%): 37.285068

Regularization vs. Accuracy (pre-trained)



```
Note:

With cbow wordvectors:

Best regularization value: 1.86E-02
Test accuracy (%): 28.868778
```