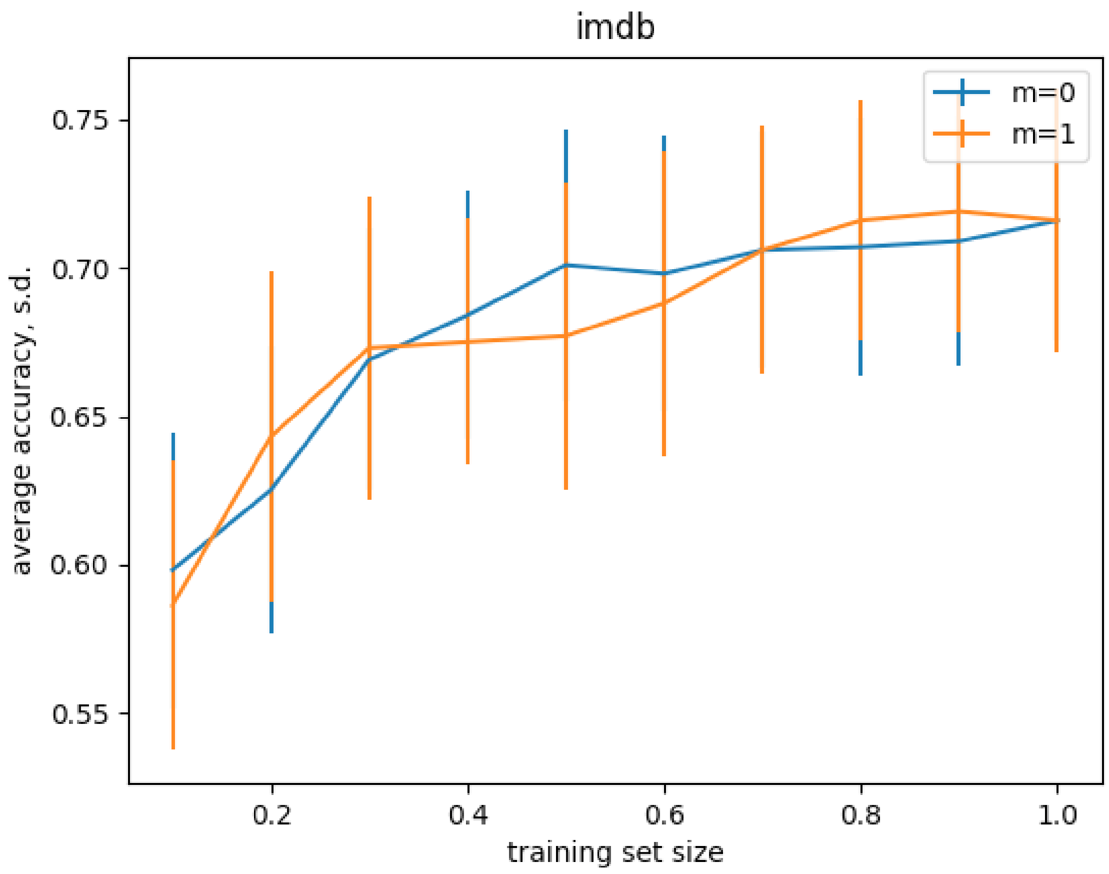
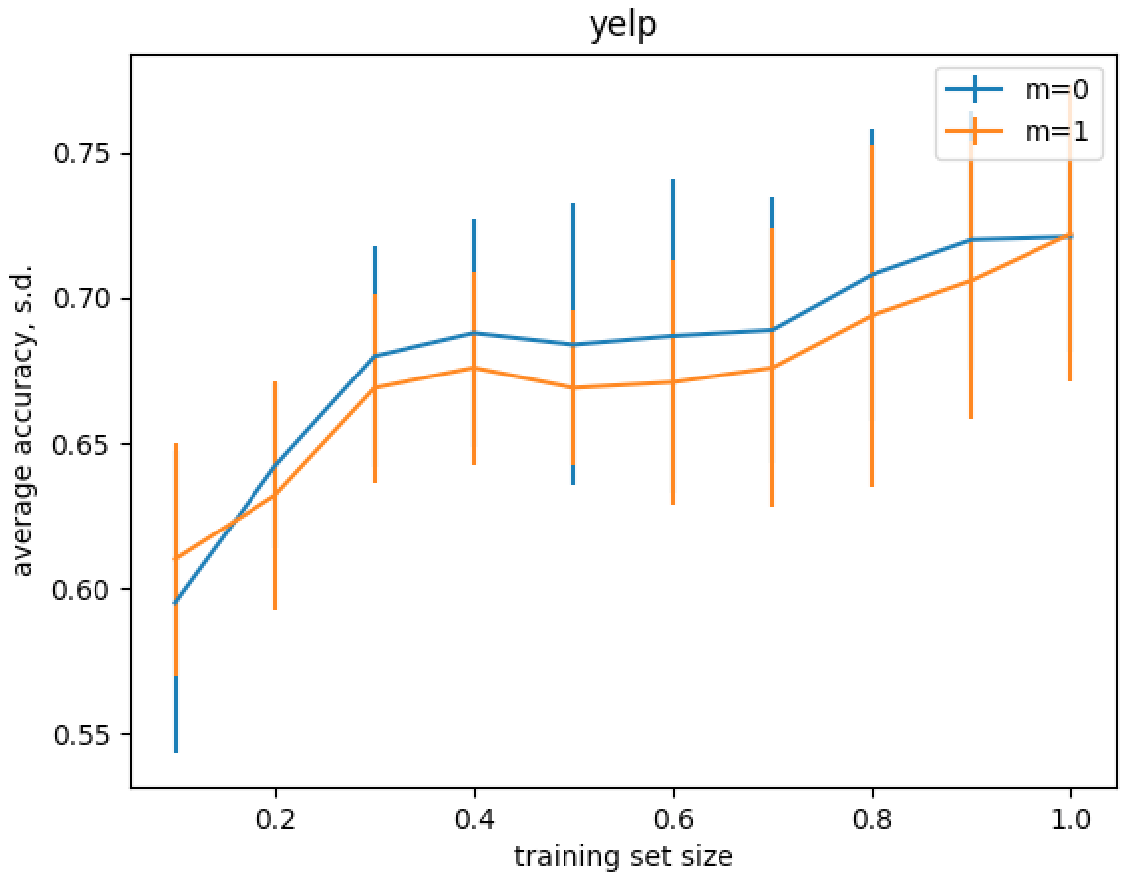
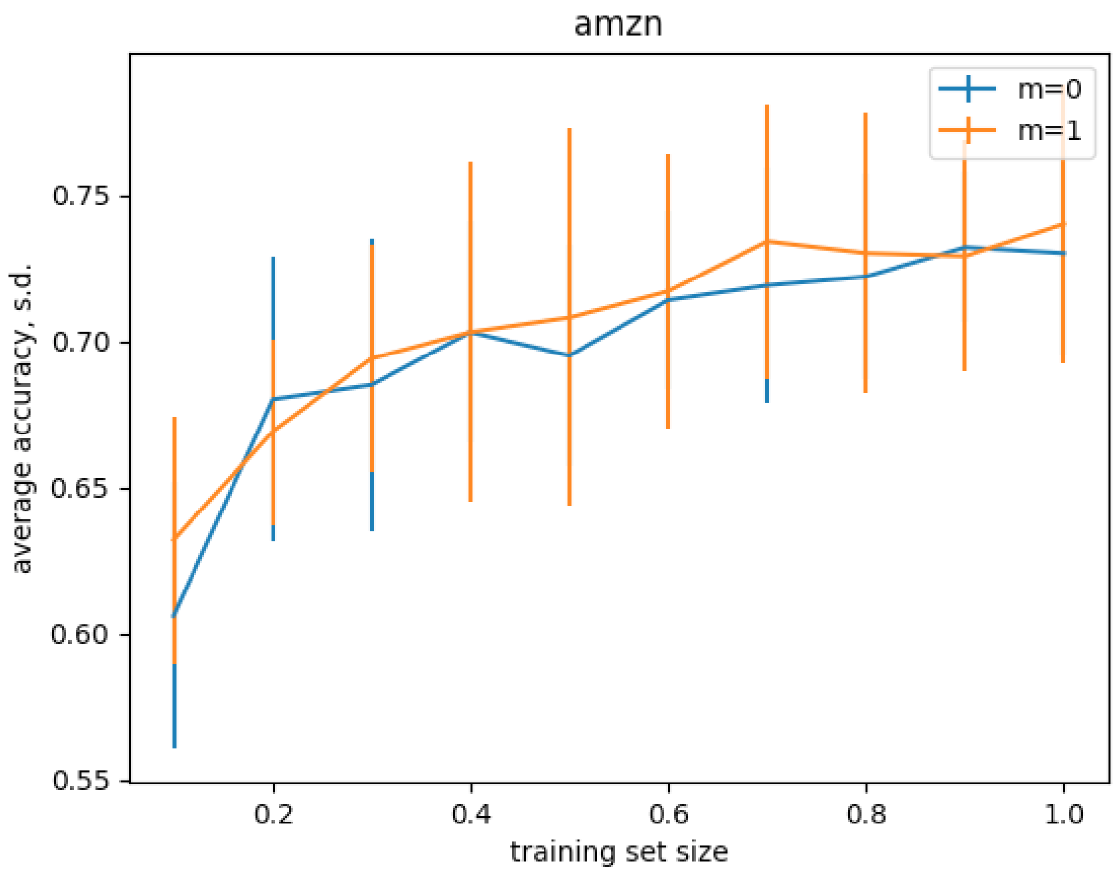
PP1 Report
Brendan McShane

Part 1

For part 1 we were testing how different training set sizes would affect accuracy and it's variance. Looking at all 3 resulting graphs (imdb, yelp, amzn), we can tell that the more data we're allowed to train on, the more accurate and reliable we were (as in we averaged a lower standard deviation across our k folds). This tends to be a trend with most machine learning models. The more data we have, the better our models tend to be to a point. Our graphs end up plateauing a little towards the end of the graph where we're using the full dataset, which implies that each individual extra data point gets less and less useful the more data we already have. Both maximum likelihood and the MAP approach performed similarly, although it would appear that MAP performs a little bit better with limited data. This makes sense with respect to both what we've discussed in class (maxL's tendency to overfit on insufficient data) and the smoothing process mentioned in the homework pdf, which helps us avoid 0/1 extreme solutions.

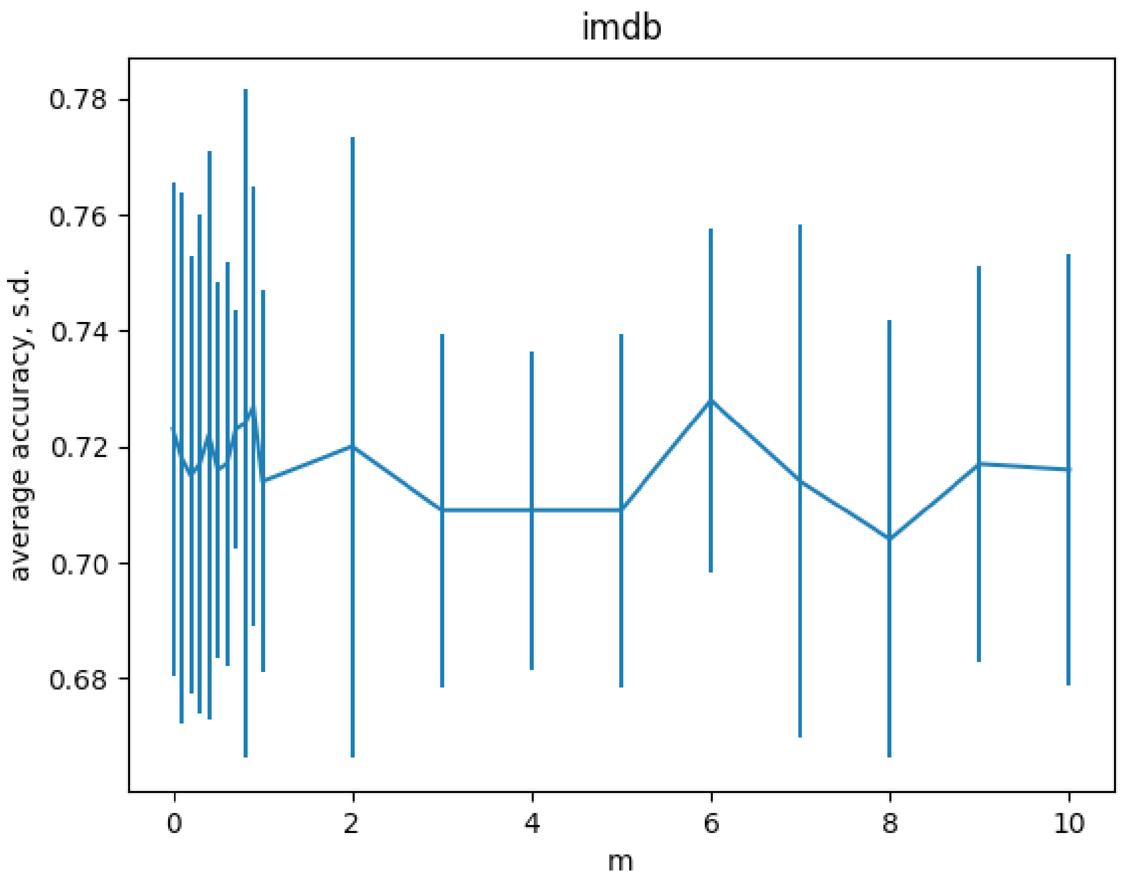


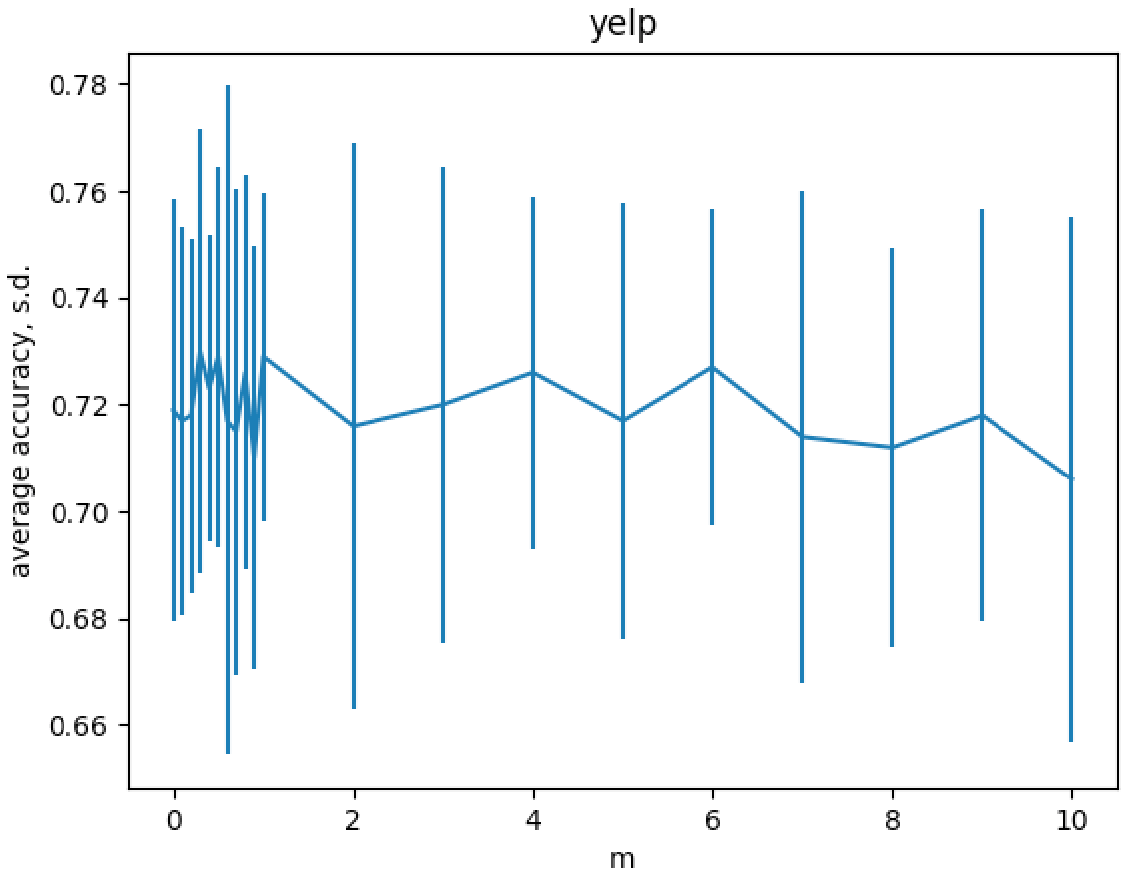

yelp
m=0
m=1
average accuracy, s.d.
0.55
0.60
0.65
0.70
0.75
0.2
0.4
0.6
0.8
1.0
training set size

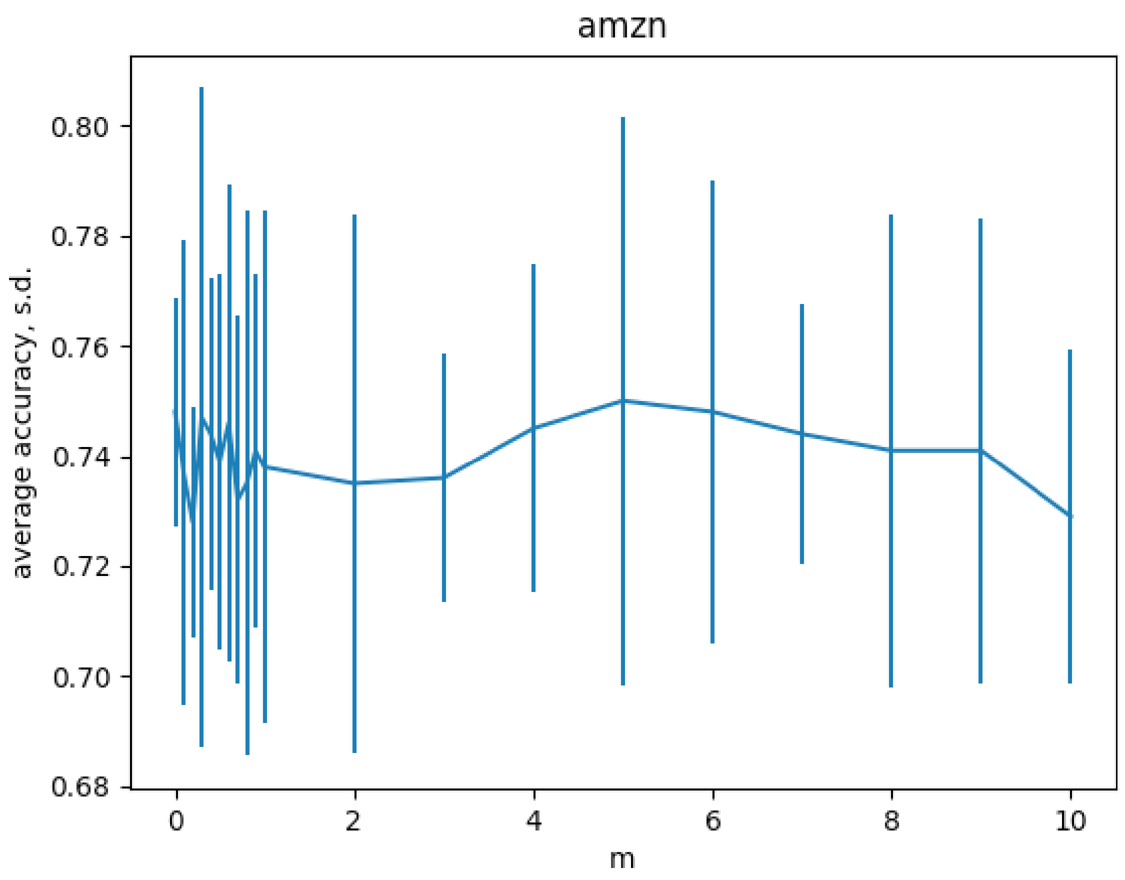

amzn
m=0
m=1
average accuracy, s.d.
0.55
0.60
0.65
0.70
0.75
0.2
0.4
0.6
0.8
1.0
training set size

Part 2

To be frank, I'm not sure what to make of these graphs. It would appear that the smaller the m value is, the more volatile our results are (notice how much the accuracies and s.d.'s are jumping all over the place in the m = [0,1] range). This makes sense to me as the entire point of the m-value (which behaves much like pseudocounts) is to smooth out our maximum likelihood values and avoid 0/1 extreme solutions. The accuracy past the m = [0,1] range seems to peak about halfway in the m = [4,6] range, and then drop off. The reliability of our results also seems to be at its best in the 4-6 range as our s.d.'s appear to be smaller in the middle of the graph. This implies relying too heavily on a large m value can actually hinder our results instead of help.




yelp
average accuracy, s.d.
0.78
0.76
0.74
0.72
0.70
0.68
0.66
0
2
4
6
8
10
m



README
I don't think there's any special instructions needed for running this code. I used python 3.8.5 and used the following packages: numpy, math, random, and matplotlib.pyplot. I just executed on the command line with 'python3 a1.py' and the 6 graphs for my results would pop up one by one (you have to exit out of each one to see the next). Other than that I can't think of anything I'd need to tell you to be able to run my code and see my results.

Instead of taking 12 different screenshots to get all my code here, I just copied and pasted. I hope that works.

```
========================================
# Brendan McShane
# B555 Programming Project 1
import random
import math
import matplotlib.pyplot as plt
import numpy as np
```

```
# tokenizes our strings into lists of words
def tokenize(inputs):
    res = []

    for review in inputs:
        curr = []
        words = review.lower().split(' ') # turns string to list of words
        for word in words:
            word = word.strip(',.?!') # removes any punctuation
            if len(word) > 0 and word[0].isalpha(): # makes sure we're not counting ' ' or '&' etc.
                curr.append(word)

        res.append(curr) # append current review in tokenized form

    return res # return list of tokenized reviews


# takes inputs and labels and returns two dictionaries of counts given class
def get_counts(inputs, labels):
    counts_given_pos = {} # {'cat': 10} = cat appears 10 times in positive reviews
    counts_given_neg = {}

    for ls, label in zip(inputs, labels): # pairs list of words to score 0 or 1
        for word in ls:
            if label==0:
                if word in counts_given_neg.keys():
                    counts_given_neg[word]+=1
                else:
                    counts_given_neg[word]=1
            else:
                if word in counts_given_pos.keys():
                    counts_given_pos[word]+=1
                else:
                    counts_given_pos[word]=1

    # returns the two dictionaries
    return counts_given_neg, counts_given_pos


# MAXIMUM LIKELIHOOD AND MAP ESTIMATES
def max_likelihood(review, neg_counts, pos_counts):
    vocab = [str(key) for key in neg_counts.keys()] + [str(key) for key in pos_counts.keys()]
    total_neg_words = sum(neg_counts.values())
    total_pos_words = sum(pos_counts.values())
    p_neg = 0
    p_pos = 0

    # summation of the class conditional log probabilities
    for word in review:
        # ignore word if not in training set
        if word in vocab:
            try:
                neg_temp = neg_counts[word]/total_neg_words
                p_neg += math.log(neg_temp)
            except:
                # in this situation neg_counts[word] threw an error because #(w^c)=0 and
neg_counts[word] doesnt exist
                p_neg += -math.inf

            try:
                pos_temp = pos_counts[word]/total_pos_words
                p_pos += math.log(pos_temp)
            except:
                p_pos += -math.inf

    # ties can go either way and dont really matter
    if p_neg > p_pos:
        return 0
    else:
        return 1


# very similar to maxL just different neg_temp/pos_temp formulas
def MAP(review, m, neg_counts, pos_counts):
    vocab = [str(key) for key in neg_counts.keys()] + [str(key) for key in pos_counts.keys()]
    V = len(vocab)
    total_neg_words = sum(neg_counts.values())
    total_pos_words = sum(pos_counts.values())
    p_neg = 0
    p_pos = 0


    for word in review:
        # ignore if word isn't in training set at all
        if word in vocab:
            # the try catch is for when #(w^c) is zero
            try:
                neg_temp = (neg_counts[word]+m)/(total_neg_words + m*V)
                p_neg += math.log(neg_temp)
            except:
                # in this situation #(w^c) = 0 and would cause an error in neg_counts[word]
                p_neg += -math.inf

            try:
                pos_temp = (pos_counts[word]+m)/(total_pos_words + m*V)
                p_pos += math.log(pos_temp)
            except:
                p_pos += -math.inf

    if p_neg > p_pos:
        return 0
    else:
        return 1


# K-FOLD CROSS VALIDATION
# we are splitting the data into k folds, and k times we're selecting a different fold to be our
testing
# data and the other k-1 folds to be our training data
def get_indexes(labels, k):
    # there is exactly a 50/50 split of positive and negative reviews
    pos_indexes = []
    neg_indexes = []

    # sorting indexes into negative and positive (stratified)
    for i in range(len(labels)):
        if labels[i]==0:
            neg_indexes.append(i)
        else:
            pos_indexes.append(i)

    # initialize k folds
    folds = []
    for i in range(k):
        folds.append([])

    # randomize our indexes
    random.shuffle(pos_indexes)
    random.shuffle(neg_indexes)

    # for every pos,neg index pair, categorize into fold 1, then the next pair into fold 2, etc..
    fold = 0
    for i in range(len(pos_indexes)):
        if fold == k:
            fold = 0

        folds[fold].append(pos_indexes[i])
        folds[fold].append(neg_indexes[i])

        fold+=1


    return folds


# maxL and MAP take a single input and returns a single prediction
# if m=0 we know we're performing maximum likelihood
def predict(inputs, neg_counts, pos_counts, m):
    yhat = []

    for review in inputs:
        if m==0:
            yhat.append(max_likelihood(review, neg_counts, pos_counts))
        else:
            yhat.append(MAP(review, m, neg_counts, pos_counts))

    # returns our list of predictions st len(yhat) = len(inputs)
    return yhat

# pretty straightforward, returns the accuracy for a yhat, y pair
def analysis(predictions, labels):
    accuracy = 0
    for y_hat, y in zip(predictions, labels):
        if y_hat == y:
            accuracy+=1

    accuracy = accuracy/len(predictions)


    return accuracy


# the meat and potatoes
def stratified_cross_val(inputs, labels, k, m, training_sizes=[]):
    inputs = tokenize(inputs)
    folds = get_indexes(labels, k)
```

```
random.shuffle(folds)

# essentially checks if we're doing experiment 1 or experiment 2
if training_sizes:
    total_results = []
    for training_size in training_sizes:
        accuracies = []

        # loops through each variation of the k folds
        for i in range(k):
            test_i = []
            train_i = []

            # separates our training and testing data
            for j in range(len(folds)):
                if j == i:
                    test_i = folds[j]
                else:
                    train_i += folds[j]

            # adjusts training data to our current desired training test size
            temp_train_i = train_i[:int(len(train_i)*training_size)]
            temp_inputs = [inputs[x] for x in temp_train_i]
            temp_labels = [labels[x] for x in temp_train_i]

            # gets class conditional word counts
            counts_given_neg, counts_given_pos = get_counts(temp_inputs, temp_labels)

            # prepares testing data
            test_inputs = [inputs[x] for x in test_i]
            test_labels = [labels[x] for x in test_i]

            # returns a yhat vector for our testing inputs
            predictions = predict(test_inputs, counts_given_neg, counts_given_pos, m)

            # records our performance
            accuracies.append(analysis(predictions, test_labels))

        # gets the average accuracy and our s.d. and records the results
        avg_acc = sum(accuracies)/len(accuracies)
        sd = np.std(accuracies)
        res = (avg_acc, sd)
        total_results.append(res)
        return total_results
    else:
        # this is all identical to the above code except for the training size loop
        # and we're returning a tuple instead of a list of tuples
        accuracies = []
        for i in range(k):
            test_i = []
            train_i = []

            for j in range(len(folds)):
                if j == i:
                    test_i = folds[j]
                else:
                    train_i += folds[j]

            temp_inputs = [inputs[x] for x in train_i]
            temp_labels = [labels[x] for x in train_i]

            counts_given_neg, counts_given_pos = get_counts(temp_inputs, temp_labels)

            test_inputs = [inputs[x] for x in test_i]
            test_labels = [labels[x] for x in test_i]

            predictions = predict(test_inputs, counts_given_neg, counts_given_pos, m)

            accuracies.append(analysis(predictions, test_labels))

        avg_acc = sum(accuracies)/len(accuracies)
        sd = np.std(accuracies)
        res = (avg_acc, sd)
        return res


# fetching datasets
imdb_inputs, imdb_labels = [], []
for review in open("pp1data/imdb_labelled.txt").readlines():
    imdb_inputs.append(review.split('\t')[0])
    imdb_labels.append(int(review.split('\t')[1].strip()))

yelp_inputs, yelp_labels = [], []
for review in open("pp1data/yelp_labelled.txt").readlines():
    yelp_inputs.append(review.split('\t')[0])
    yelp_labels.append(int(review.split('\t')[1].strip()))
```

```
amzn_inputs, amzn_labels = [], []
for review in open("pp1data/amazon_cells_labelled.txt").readlines():
    amzn_inputs.append(review.split('\t')[0])
    amzn_labels.append(int(review.split('\t')[1].strip()))



# bulletpoint 1
# aggregating our results
sets = [(imdb_inputs, imdb_labels), (yelp_inputs, yelp_labels), (amzn_inputs, amzn_labels)]

training_sizes = [.1,.2,.3,.4,.5,.6,.7,.8,.9,1]
ms = [0,1]
results_by_set = []
# loops through all 3 datasets
for set in sets:
    inputs = set[0]
    labels = set[1]

    results_by_m = []
    # loops through m=0 and m=1
    for m in ms:
        accXts = stratified_cross_val(inputs, labels, 10, m, training_sizes)
        results_by_m.append(accXts)

    results_by_set.append(results_by_m)


# the end result data for the above code is essentially a 3x1 matrix for each dataset,
# where each member of the matrix is a 10x1 matrix representing the 10 training set sizes by
(avg_acc, s.d.)

# graphing our results
training_sets = ['imdb','yelp','amzn']
for i in range(len(training_sets)):
    accuracies_m0 = [tup[0] for tup in results_by_set[i][0]]
    sd_m0 = [tup[1] for tup in results_by_set[i][0]]
    accuracies_m1 = [tup[0] for tup in results_by_set[i][1]]
    sd_m1 = [tup[1] for tup in results_by_set[i][1]]

    plt.errorbar(y=accuracies_m0, x=training_sizes, yerr=sd_m0, label='m=0')
    plt.errorbar(y=accuracies_m1, x=training_sizes, yerr=sd_m1, label='m=1')
    plt.xlabel('training set size')
    plt.ylabel('average accuracy, s.d.')
    plt.legend()
    plt.title(training_sets[i])
    plt.show()



# bulletpoint 2
# aggregrating our results
ms = [0,.1,.2,.3,.4,.5,.6,.7,.8,.9,1,2,3,4,5,6,7,8,9,10]
results_by_set = []
for set in sets:
    inputs = set[0]
    labels = set[1]

    results_by_m = []
    for m in ms:
        results_by_m.append(stratified_cross_val(inputs, labels, 10, m))

    results_by_set.append(results_by_m)


# the end result data for the above code is a 3x1 matrix where each member is
# a 20x1 matrix representing the (avg_acc, s.d.) for all 20 m values


sets = ['imdb', 'yelp', 'amzn']
for i in range(len(sets)):
    plt.title(sets[i])

    accuracies = [tup[0] for tup in results_by_set[i]]
    sd = [tup[1] for tup in results_by_set[i]]

    plt.errorbar(y=accuracies, x=ms, yerr=sd)
    plt.xlabel('m')
    plt.ylabel('average accuracy, s.d.')
    plt.title(sets[i])
    plt.show()
```