6.864 PSET1

Part 1

Implementation Details: In this part, we explore different ways of using unlabeled text to learn word representations. Using a dataset on product review, we first created a term document matrix and implemented a function to compute word representations using LSA. The latter was done using the TruncatedSVD function from sklearn.decomposition library. We then implemented the TF-IDF transform where we multiply each entry in the term document matrix by its inverse document frequency. Lastly, we train a logistic regression model on a set of labeled reviews using three different featurizers. The LSA featurizer contains the learned feature representation of each review, which is calculated by summing LSA word representations. We examined how much representations learned from unlabeled reviews improve revsiew classification.

Question 1. The nearest neighbors in representation space are somehow semantically related to each other. For example, the most similar words to "the" are: "of", "and", and "to"; these words are often used in various sentences and do not contribute significantly to the meaning of the sentence.

The most similar words to "dog" are: "food", "pet", "switched", and "foods"; each of these words has a somewhat direct connection to "dog" except for "switched".

The most similar words to "good" are: ".", "a", "but", ",", and "the"; these words do not have apparent connection to "good" in terms of meaning. However, they are often in proximity to "good". For example, "a good person", "the good food", or "I am good.".

Question 2. The size of the LSA representations affects this behavior because it indicates the number of latent space we are mapping each word to. If the LSA representation is too small, very little latent information of the words will be learned. However, if the LSA representations is too large, the embedding matrix might contain too much irrelevant information that can be misleading when mapping words to latent space (overfitting). When we decrease the size of the LSA representation to 5, some words produce poor results. For example, the most similar words to "dog" are: "him", "food", "formula", "general", and "began". These words have weaker connection to "dog" than the words generated in Question 1 with LSA representation size of 500. However, some words such as "the" continue to produce good results; the most similar words to "the" are: "a", "be", "need", "but", and ".". This might be due to the fact that these words also lie close to the most represented latent dimension. Thus, decreasing the number of latent dimensions does not influence their nearest neighbors significantly.

Question 3. Let $W_{td} = U\Sigma V^T$, then $W_{tt} = W_{td}W_{td}^T = U\Sigma V^T V\Sigma^T U^T$. Since U and V are both orthogonal matrices given the properties SVD, $V^T V$ will result in an identity matrix. Then, we can simplify W_{tt} to $U\Sigma\Sigma^T U^T$. Since Σ is diagonal, Σ^T will also be diagonal where each element is the square of the corresponding element in . Let $\Sigma_{tt} = \Sigma\Sigma^T$, $W_{tt} = U\Sigma_{tt}U^T$. Therefore, the left singular vectors of W_{td} and W_{tt} are the same. However, we can do not fully observe this behavior in our implementation. As shown in Fig. 1 and Fig. 2, most of the word obtain the different similar words although in both cases, they still fit well together with our target words. These discrepancies are due to the numerical estimations when performing SVD.

Word	Similar Words
"good"	"," "but" "a" "and" "is"
"bad"	"." "taste" "but" "a" "not"
"cookie"	"nana's" "cookies" "bars" "oreos" "moist"
"jelly"	"twist" "cardboard" "advertised" "plum" "sold"
"dog"	"food" "pets" "pet" "foods" "switched"
"the"	"." "and" "<unk>" "of" "to"
"4"	"1" "6" "70" "stevia" "5"

Fig. 1. Similar words after performing LSA on term-document matrix

Word	Similar Words
"good"	"better" "but" "too" "like" "really"
"bad"	"like" "taste" "just" "really" "better"
"cookie"	"a" "cookies" "amazon" "whole" "don't"
"jelly"	"." "bought" "came" "was" "found"
"dog"	"dogs" "food" "pet" "vet" "years"
"the"	"on" "only" "of" "at" "same"
"4"	"1" "6" "2" "5" "3"

Fig. 2. Similar words after performing LSA on co-occurrence matrix

Question 4. Learned representations alone do not help with the review classification problem in most cases. This is because the many reviews, positive reviews will usually contain such positive terms as "good", "like", and "better" more frequently. However, the LSA featurizer does not account for the frequency of terms. To make the matter worse, LSA suggests that "bad" is very similar to "like" and "better" as shown in Fig. 2. As a result, such word as "like" and "better" are treated as signals for both positive and negative reviews, contributing to their poor performance.

However, in some cases, the combo featurizer performs better than both the word and the LSA featurizer. This is because the combo featurizer contains information on both the frequency of terms and their latent embeddings.

As the number of labeled examples increases, the word embeddings increases the accuracy for LSA featurizer. This is because the word embeddings become more precise as the number of examples increase. However, the accuracy for lsa, and for word and combo, all plateau at around 1,000 exam-

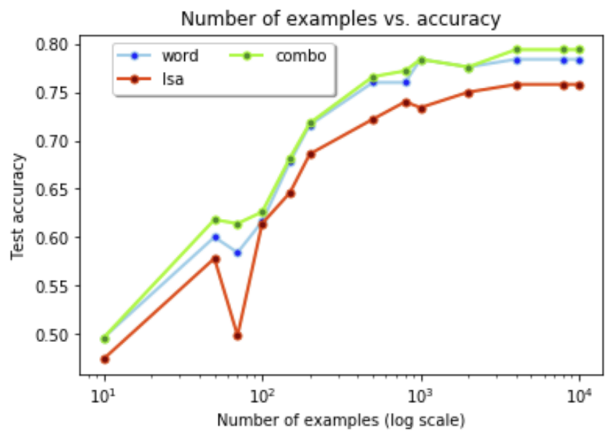


Fig. 3. Test accuracy of word, lsa, and combo for various number of examples with 500 LSA representation size

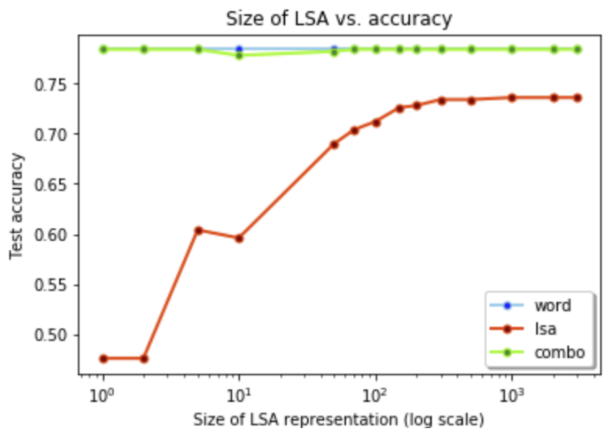


Fig. 4. Test accuracy of word, lsa, and combo for various LSA representation size with 1000 examples

ples. After this point, additional examples will not increase accuracy significantly.

Question 5. As the size of word embeddings increases, the accuracy of the lsa featurizer also increases because the higher latent space is able to encode more representations for each word. However, when the size of the word embeddings reaches around 100, the accuracy of the lsa featurizer begins to reach its peak as shown in Fig. 4. After this peak, the accuracy drops subtly because the word embeddings might be overfitting due to the high representation size.

One interesting trend in Fig. 4 is that the accuracy of combo remains relatively high and unchanged throughout. Its accuracy is quite high even when the accuracy of the lsa featurizer is significantly lower. Thus, it can be inferred that that the combo featurizer relies more on the word featurizer than the lsa featurizer.

Part 2

Implementation Details: In Part 2, we trained a word embedding model with a word2vec-style objective rather than a matrix factorization objective. We wrote a torch module where the forward function takes a batch of context word ID's and predict the word in the middle of the context, as in the CBOW moded. This was done by adding embeddings, a hidden linear layer, and a softmax output that returns the most probable word. The training function takes in a corpus of training sentences and returns a matrix of word embeddings with the same structure as Part 1. We used the Adam optimizer with a learning rate of 0.001. We used cross entropy loss as our loss function.

After training the word embeddings, we visualized the embedding space and check its closest neighbors. We also ran clustering and qualitatviely looked for underlying pattern in clusters. Lastly, we used the trained word embeddings to constructor vector representations of full reviews by simply averaging all the word embeddings in the review to create an overall embedding. We then evaluated their accuracy in the same review classification task from Part 1.

Question 1. The nearest neighbors in representation space are much more similar than those from Part 1. Qualitatively, most of these words represents similar words more naturally in product reviews.

For example, the most similar words to "the" are: "my", "a", "their", "your", "this"; like in Part 1, words similar to "the" are often used in various sentences and do not contribute significantly to the meaning of the sentence. However, similar words in this case are more related and are usually followed by nouns.

The most similar words to "dog" are: "puppy", "snacks", "pouch", "cafe", and "pet"; each of these words has more apparent direct connections to "dog" than those in part 1. Lastly, the most similar words to "good" are: "restaurant", "excited", "difficult", "great", and "superior". In Part 1, similar words to "good" do not have apparent connection to "good" in terms of meaning, but they are often in proximity to "good". On the other hand, in this scenario, the most similar words to "good" have apparent connection in terms of meaning.

Question 2. Qualitatively, increasing the context size produces more similar words. For example, when the context size is 1, the most similar words of "dog" are: "cat", "sea", "condition", "prepared", and "belgian"; most of which do not have direct connection to "dog". However, as we increase the context size to 5, the most similar words of "dog" are: "p", "sweeteners", "pets", "co", and "live"; most of which have some connection to "dog". Increasing the context size allows the model to account to take more words in proximity into consideration. However, if the context size becomes too large, accuracy decreases. As shown in Fig. 5, increasing the context size first improves accuracy. When the context size becomes around 10, the accuracy decreases significantly. This is because words outside the context range of the target word are taken into consideration when learning embeddings for our target word.

Question 3. The learned word embeddings produce higher accuracy with fewer training examples. As shown in Fig. 6, at around 1000 examples, the learned word embeddings reached an accuracy of around 0.80. On the other hand, as shown in

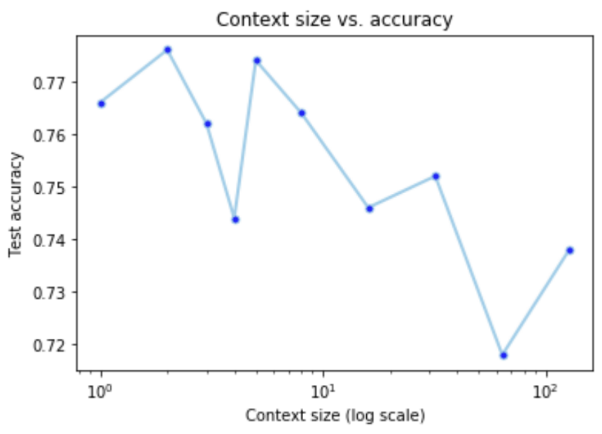


Fig. 5. Test accuracy of various context sizes trained with 500 representation size, 10 epoches, and 100 batch size. Vector representations of reviews computed by averaging all word embeddings in the review

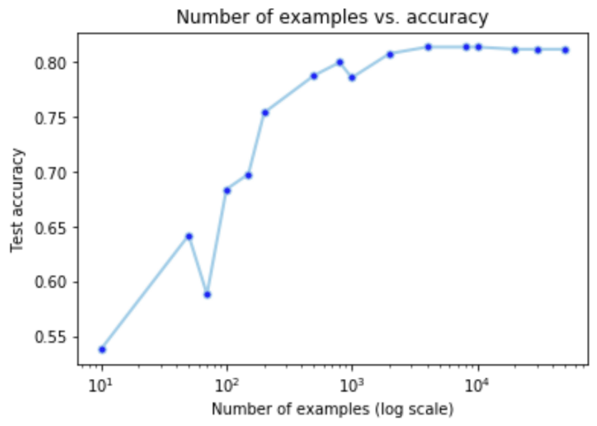


Fig. 6. Test accuracy of different number of examples trained with 500 representation size, 30 epoches, and 100 batch size. Vector representations of reviews computed by averaging all word embeddings in the review

Fig. 3, the best featurizer only reached an accuracy of around 0.75.

The learned word embeddings reaches its highest accuracy of around 0.83 at around 10000 examples. The accuracy then begins to decrease due to overfitting. On the other hand, the best featurizer in Fig. 3 is only able to reach a highest accuracy of around 0.78 at around 10000 examples.

Another interesting trend in both figures is that in both graphs, the accuracy experience a sharp dip right before 100 examples. We were not able to figure out the cause of this significant dip.

Question 4. One advantage of using the CBOW model is that it considers words in proximity when learning representations. On the other hand, LSA is a bag-of-words model that mainly relies on the term document frequencies to generate word embeddings without considering the order of words. As a result, CBOW model is able to produce higher accuracy, better representation, even with fewer training examples.

Some disadvantages of using the CBOW model is that it has a long training time and can possibly overfit. However, another problem that both CBOW and LSA do not address is identifying embeddings for words with multiple meanings since they only have one vector representation for each word.

Question 5. One problem with constructing a representation of the review by averaging the embeddings of the individual words is that sentences with double negatives might seem much worse than expected because averaging the embeddings does not take the words in context into consideration. For example, the review "It wasn't bad at all, I enjoyed it." can be classified as neutral due to the double negative in the first part of the sentence while it should be classified positive.

Part 3

Implementation Details: In this part, we use the Baum-Welch algorithm to learn categorical representations of words in the vocabulary. First, we implemented the forward-backward algorithm for HMM as we've seen in lecture. To prevent underflowing, most all computation was done in log space. More specifically, $log(ab) = log(a) + log(b)$; and if $x = log(a)$ and $y = log(b)$, $log(a+b) = logaddexp(x, y)$. In many scenarios, we must perform logaddexp() across the rows or columns of a matrix. To vectorize computation, we used logsumexp() from scipy.special library. After having computed $\alpha_t(j)$ and $\beta_t(i)$, we calculated $\xi_t(i,j)$ and $\gamma_t(i)$. At the end of each iteration of Baum-Welch, we update A, B, and pi using ξ and γ.

Note that in our implementation of forward-backward, we leave A, B, and pi in standard decimal space. However, α, β, γ, and ξ are all in log space. Additionally, as we loop through each review in corpus, using ξ and γ, we calculate a running sum of $E[s_i \to s_j]$, $E[s_i \to s_*]$, $E[s_j, w_k]$, $E[s_j]$, and $E[q_1 = s_i]$. At the end of each iteration, we also divided each term in $E[q_1 = s_i]$ by the length of the corpus. We did not divide other expected values by the length of the corpus because they would cancel each other out when updating A, B, and π.

We then repeated classification experiment from Parts 1 and 2 using the vector of expected hidden state counts as a sentence representation.

Question 1. We trained the model with two hidden states for three iterations, since the total log probability converges at around three iterations. The first state contains names of objects such as "plum", "mints", and "plocky's". The second state contains words that have weak indication about quality of the product such as "remaining", "risk", "burn", "improve", "harder", "rip", and etc.

We trained the model with 10 hidden states for three iterations. Each of the ten state contains words that are somehow related; however, there isn't a clear boundary between each hidden state. For example, state 1 contains "flavour", "restaurant", "ordering", "oven", and "asked"; all these words are related to food and restaurant. State 3 contains "earlier", "target", "serious", and "mushy"; these words tend to deal with the condition and state of the product. State 7 contains "sad", "paying", "offer", and "handy"; these words have indirect connections to payment. Most of the states contain words describing the various aspects of a product; and these aspects

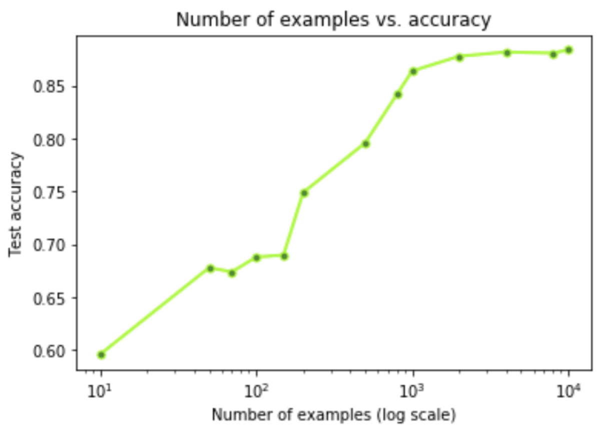


Fig. 7. Test accuracy of different number of examples. Embeddings are learned using the Baum-Welch algorithm with 5 iterations.

(payment, condition, quality, state, and etc.) are often linked to whether the product receives a positive or negative review.

We started training the model with 100 hidden states but did not finish due to long training time. However, using the pattern from two states and 10 states, we suppose that each of the 100 states will contain more detailed description that contribute to product review. These detailed descriptions can include price, arrival time, portion, service, and etc.

Question 2. As the number of labeled examples increases, HMM-based sentence representations achieve higher accuracy on the same review classification task from Parts 1 and 2 of the homework. As shown in Fig. 7, the HMM-based sentence representations achieve a accuracy higher than 0.85 at 1000 examples. As the number of examples continue to increase, the accuracy plateaus at around 0.90. In contrast, as shown in 3, the best featurizer from Part 1 only obtained a highest accuracy of around 0.80. As shown in 6, the embeddings from the CBOW model only obtained a highest accuracy of around 0.83. The high accuracy of the HMM-based representation is that it considers the observations in sequence, and human language is fundamentally sequential in nature. On the other hand, in Part 1, we applied lsa featurizer to generate review representations. Meanwhile, in Part 2, we simply constructed review representations by averaging the embeddings of the individual words. However, both approaches do not account for the sequential ordering of the words in the review.

The HMM state distributions are sensible to sentence representation because HMM is a probabilistic method of modeling time series data. The hidden state distributions are generated based on observations at each discrete sequential timestamps. Because human sentences are also sequential in nature, the hidden state distributions conditioned on observed words are sensible. As evident in Fig. 7, HMM-based sentence representations achieve higher accuracy then the earlier models in Parts 1 and 2.

```
%%bash
!(stat -t /usr/local/lib/*/dist-packages/google/colab > /dev/null 2>&1) && exit
rm -rf 6864-hw1
git clone https://github.com/lingo-mit/6864-hw1.git
```

```
import sys
sys.path.append("/content/6864-hw1")

import csv
import itertools as it
import numpy as np
np.random.seed(0)

import lab_util

import math
from sklearn.decomposition import TruncatedSVD
```

Introduction

In this lab, you'll explore three different ways of using unlabeled text data to learn pretrained word rep
the effects of different modeling decisions (representation learning objective, context size, etc.) on bo
representations and their effect on a downstream prediction problem.

General lab report guidelines

Homework assignments should be submitted in the form of a research report. (We'll be providing a pl
but are still sorting out some logistics.) Please upload PDFs, with a maximum of four single-spaced p
Association for Computational Linguistics style files.) Reports should have one section for each part
section should describe the details of your code implementation, and include whatever charts / table
questions at the end of the corresponding homework part.

We're going to be working with a dataset of product reviews. It looks like this:

```
data = []
n_positive = 0
n_disp = 0
with open("/content/6864-hw1/reviews.csv") as reader:
  csvreader = csv.reader(reader)
  next(csvreader)
  for id, review, label in csvreader:
    label = int(label)

    # hacky class balancing
    if label == 1:
      if n_positive == 2000:
        continue
      n_positive += 1
    if len(data) == 4000:
      break

    data.append((review, label))

    if n_disp > 5:
      continue
    n_disp += 1
    print("review:", review)
    print("rating:", label, "(good)" if label == 1 else "(bad)")
    print()

print(f"Read {len(data)} total reviews.")
np.random.shuffle(data)
reviews, labels = zip(*data)
train_reviews = reviews[:3000]
train_labels = labels[:3000]
val_reviews = reviews[3000:3500]
val_labels = labels[3000:3500]
test_reviews = reviews[3500:]
test_labels = labels[3500:]
```

We've provided a little bit of helper code for reading in the dataset; your job is to implement the learnii

Part 1: word representations via matrix factorization

First, we'll construct the term--document matrix (look at `/content/6864-hw1/lab_util.py` in the fil
how this works).

```
vectorizer = lab_util.CountVectorizer()
vectorizer.fit(train_reviews)
td_matrix = vectorizer.transform(train_reviews).T
print(f"TD matrix is {td_matrix.shape[0]} x {td_matrix.shape[1]}")
```

First, implement a function that computes word representations via latent semantic analysis:

```

def learn_reps_lsa(matrix, rep_size):
  # `matrix` is a `|V| x n` matrix, where `|V|` is the number of words in the
  # vocabulary. This function should return a `|V| x rep_size` matrix with each
  # row corresponding to a word representation. The `sklearn.decomposition`
  # package may be useful.

  # Your code here!
  svd = TruncatedSVD(n_components=rep_size, n_iter=30, random_state=42)
  svd.fit(matrix)
  return svd.transform(matrix)
```

link textLet's look at some representations:

```
reps = learn_reps_lsa(td_matrix.dot(td_matrix.T), 1000)
words = ["good", "bad", "cookie", "jelly", "dog", "the", "4"]
show_tokens = [vectorizer.tokenizer.word_to_token[word] for word in words]
lab_util.show_similar_words(vectorizer.tokenizer, reps, show_tokens)
```

We've been operating on the raw count matrix, but in class we discussed several reweighting scheme more informative.

Here, implement the TF-IDF transform and see how it affects learned representations.

```
def transform_tfidf(matrix):
  matrix_copy = matrix.copy()
  matrix_copy[matrix_copy>0.]=1.
  # print(len(matrix_copy[matrix_copy == 0])+len(matrix_copy[matrix_copy==1]))

  num_docs_containing_word = np.sum(matrix_copy, axis = 1)
  # print(num_docs_containing_word)
  for i in range(len(matrix)):
    for j in range(len(matrix[0])):
      matrix[i][j]*=math.log(len(matrix[0])/num_docs_containing_word[i],10)
  return matrix
  # matrix is a |V| x |D| matrix of raw counts, where |V| is the
  # vocabulary size and |D| is the number of documents in the corpus. This
  # function should (nondestructively) return a version of matrix with the
  # TF-IDF transform appliied.

  # Your code here!
```

How does this change the learned similarity function?

```
td_matrix_tfidf = transform_tfidf(td_matrix)# dot(td_matrix.T)
td_matrix_tfidf = transform_tfidf(td_matrix)#.dot(td_matrix.T)
reps_tfidf = learn_reps_lsa(td_matrix_tfidf, 500)
lab_util.show_similar_words(vectorizer.tokenizer, reps_tfidf, show_tokens)
```

Now that we have some representations, let's see if we can do something useful with them.

Below, implement a feature function that represents a document as the sum of its learned word embe

The remaining code trains a logistic regression model on a set of *labeled* reviews; we're interested in s
from *unlabeled* reviews improve classification.

```
def word_featurizer(xs):
  # normalize
  return xs / np.sqrt((xs ** 2).sum(axis=1, keepdims=True))

def lsa_featurizer(xs):
  # This function takes in a matrix in which each row contains the word counts
  # for the given review. It should return a matrix in which each row contains
  # the learned feature representation of each review (e.g. the sum of LSA
  # word representations).

  feats = np.dot(xs, reps_tfidf)
  # feats = None # Your code here!
  # normalize
  return feats / np.sqrt((feats ** 2).sum(axis=1, keepdims=True))

def combo_featurizer(xs):
  return np.concatenate((word_featurizer(xs), lsa_featurizer(xs)), axis=1)

def train_model(featurizer, xs, ys):
  import sklearn.linear_model
  xs_featurized = featurizer(xs)
  model = sklearn.linear_model.LogisticRegression()
  model.fit(xs_featurized, ys)
  return model

def eval_model(model, featurizer, xs, ys):
  xs_featurized = featurizer(xs)
  pred_ys = model.predict(xs_featurized)
  print("test accuracy", np.mean(pred_ys == ys))

def training_experiment(name, featurizer, n_train):
  print(f"{name} features, {n_train} examples")
  train_xs = vectorizer.transform(train_reviews[:n_train])
  train_ys = train_labels[:n_train]
  test_xs = vectorizer.transform(test_reviews)
  test_ys = test_labels
  model = train_model(featurizer, train_xs, train_ys)
  eval_model(model, featurizer, test_xs, test_ys)
  print()

training_experiment("word", word_featurizer, 500)
training_experiment("lsa", lsa_featurizer, 500)
training_experiment("combo", combo_featurizer, 500)
```

Part 1: Lab writeup

Part 1 of your lab report should discuss any implementation details that were important to filling out t
up experiments that answer the following questions:

1. Qualitatively, what do you observe about nearest neighbors in representation space? (E.g. what
   good?)
2. How does the size of the LSA representation affect this behavior?
3. Recall that the we can compute the word co-occurrence matrix $W_{tt} = W_{td}W_{td}^\top$. What can you
   left singular vectors of W_{td} and W_{tt}? Do you observe this behavior with your implementation of
4. Do learned representations help with the review classification problem? What is the relationship
   and the effect of word embeddings?
5. What is the relationship between the size of the word embeddings and their usefulness for the c

Part 2: word representations via language modeling

In this section, we'll train a word embedding model with a word2vec-style objective rather than a matr
little more work; we've provided scaffolding for a PyTorch model implementation below. (If you've nev
tutorials here. You're also welcome to implement these experiments in any other framework of your c

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data as torch_data

class Word2VecModel(nn.Module):
  # A torch module implementing a word2vec predictor. The `forward` function
  # should take a batch of context word ids as input and predict the word
  # in the middle of the context as output, as in the CBOW model from lecture.

  def __init__(self, vocab_size, embed_dim):
      super().__init__()
      self.embeddings = nn.Embedding(vocab_size, embed_dim)
      self.linear = nn.Linear(embed_dim, vocab_size)
      # Your code here!


  def forward(self, context):
      # Context is an `n_batch x n_context` matrix of integer word ids
      # this function should return a set of scores for predicting the word
      # in the middle of the context
```

```

    # Your code here!
    embeddings = self.embeddings(context)
    embed_sum = embeddings.sum(dim=1)
    hidden = self.linear(embed_sum)
    output = F.log_softmax(hidden, dim=1)
    return output


```

```
def learn_reps_word2vec(corpus, window_size, rep_size, n_epochs, n_batch):
  # This method takes in a corpus of training sentences. It returns a matrix of
  # word embeddings with the same structure as used in the previous section of
  # the assignment. (You can extract this matrix from the parameters of the
  # Word2VecModel.)

  tokenizer = lab_util.Tokenizer()
  tokenizer.fit(corpus)
  tokenized_corpus = tokenizer.tokenize(corpus)

  ngrams = lab_util.get_ngrams(tokenized_corpus, window_size)

  device = torch.device('cuda')  # run on colab gpu
  model = Word2VecModel(tokenizer.vocab_size, rep_size).to(device)
  opt = optim.Adam(model.parameters(), lr=0.001)
  loss_fn = nn.CrossEntropyLoss() # Your code here

  loader = torch_data.DataLoader(ngrams, batch_size=n_batch, shuffle=True)

  for epoch in range(n_epochs):
    for context, label in loader:
      # as described above, `context` is a batch of context word ids, and
      # `label` is a batch of predicted word labels
      model.zero_grad()
      pred = model(context.cuda()).cuda()
      label = label.cuda()
      #print(pred.shape, label.shape)
      loss = loss_fn(pred, label)
      loss.backward()
      opt.step()
      # Your code here!

  # reminder: you want to return a `vocab_size x embedding_size` numpy array
  embedding_matrix = []
  # Your code here!
  for layer in model.embeddings.parameters():
    embedding_matrix.append(layer.data.cpu().numpy())
  return embedding_matrix
```

```
# reps_word2vec = learn_reps_word2vec(train_reviews, 2, 500, 30, 100)
reps_word2vec = learn_reps_word2vec(train_reviews, 2, 500, 10, 100)
```

```
reps_word2vec = learn_reps_word2vec(train_reviews, 2, 500, 10, 100)
reps_word2vec = np.array(reps_word2vec).squeeze()
```

After training the embeddings, we can try to visualize the embedding space to see if it makes sense. F
and check its closest neighbors.

```
lab_util.show_similar_words(vectorizer.tokenizer, reps_word2vec, show_tokens)
```

```
good 47
  piece 1.683
  metallic 1.684
  french 1.711
  easy 1.715
  vanilla 1.715
bad 201
  crazy 1.691
  further 1.704
  solid 1.705
  liquid 1.713
  carried 1.715
cookie 504
  lime 1.680
  roll 1.717
  concentrated 1.728
  wrapped 1.730
  hold 1.736
jelly 351
  sucralose 1.664
  clams 1.707
  general 1.709
  commercial 1.725
  individual 1.732
dog 925
  store 1.686
  maker 1.688
  note 1.710
  caused 1.736
  r 1.741
the 36
  my 1.537
  a 1.555
  their 1.667
  another 1.691
  an 1.705
4 292
  gallon 1.700
  toddler 1.703
  six 1.719
```

We can also cluster the embedding space. Clustering in 4 or more dimensions is hard to visualize, an
because there are so many words in the vocabulary. One thing we can try to do is assign cluster label

```
from sklearn.cluster import KMeans

indices = KMeans(n_clusters=10).fit_predict(reps_word2vec)
zipped = list(zip(range(vectorizer.tokenizer.vocab_size), indices))
np.random.shuffle(zipped)
zipped = zipped[:100]
zipped = sorted(zipped, key=lambda x: x[1])
for token, cluster_idx in zipped:
  word = vectorizer.tokenizer.token_to_word[token]
  print(f"{word}: {cluster_idx}")
```

```
        bottle: 1
        unfortunately: 1
        baby: 1
        dip: 1
        dressing: 1
        horrible: 1
        stuff: 1
        test: 1
        mother: 1
        chunks: 1
        sesame: 1
        disappointed: 1
        cereal: 1
        serious: 1
        mint: 1
        figure: 1
        remember: 1
        decided: 1
        craving: 1
        claims: 1
        switch: 1
        co: 1
        state: 1
        health: 1
        touch: 1
        spinach: 1
        than: 1
        described: 1
        tasting: 2
        sunflower: 5
        lid: 5
        watching: 5
        contacted: 5
        thats: 5
        quantity: 5
        palatable: 5
        second: 5
        guests: 5
        speak: 5
        much: 5
        twist: 5
        remaining: 5
        chewy: 7
        i: 7
        potato: 7
        paste: 7
        kind: 7
        mustard: 7
        flavour: 7
        allergy: 7
        peanuts: 7
```

```
    tin: 7
    barely: 7
    chocolates: 7
    beverage: 7
    toddler: 7
    design: 7
    decaffeinated: 7
    bears: 7
    left: 7
    shape: 7
    tough: 7
    nutritious: 7
    low: 7
    beef: 7
    shows: 7
    worst: 7
    wait: 8
    trouble: 8
    gives: 9
    irish: 9
    citrus: 9
    anywhere: 9
    reduced: 9
    placed: 9
    enjoy: 9
    she: 9
    about: 9
    date: 9
    such: 9
    complete: 9
    egg: 9
    paying: 9
    puppy: 9
    become: 9
    clear: 9
    fell: 9
    write: 9
    maybe: 9
    thing: 9
    worth: 9
    today: 9
    like: 9
    lot: 9
```

Finally, we can use the trained word embeddings to construct vector representations of full reviews. (
average all the word embeddings in the review to create an overall embedding. Implement the transfo
this.

```
def lsa_featurizer(xs):
  feats = np.dot(xs, reps_word2vec)
  # This function takes in a matrix in which each row contains the word counts
  # for the given review. It should return a matrix in which each row contains
  # the learned feature representation of each review (e.g. the sum of LSA
  # word representations).

  # normalize
  return feats / np.sqrt((feats ** 2).sum(axis=1, keepdims=True))

training_experiment("word2vec", lsa_featurizer, 500)
```

```
word2vec features, 500 examples
test accuracy 0.772
```

```
# for examples in # examples = [10,50,70,100,150,200,500,800,1000,2000,4000,8000,1
  # print(examples)
reps_word2vec = learn_reps_word2vec(train_reviews, 2, 500, 30, 100)
reps_word2vec = np.array(reps_word2vec).squeeze()
for examples in [20000, 30000, 50000]:
  training_experiment("word2vec", lsa_featurizer, examples)
```

```
word2vec features, 20000 examples
test accuracy 0.812

word2vec features, 30000 examples
test accuracy 0.812

word2vec features, 50000 examples
test accuracy 0.812
```

2 word2vec features, 500 examples test accuracy 0.776

4 word2vec features, 500 examples test accuracy 0.744

8 word2vec features, 500 examples test accuracy 0.764

16 word2vec features, 500 examples test accuracy 0.746

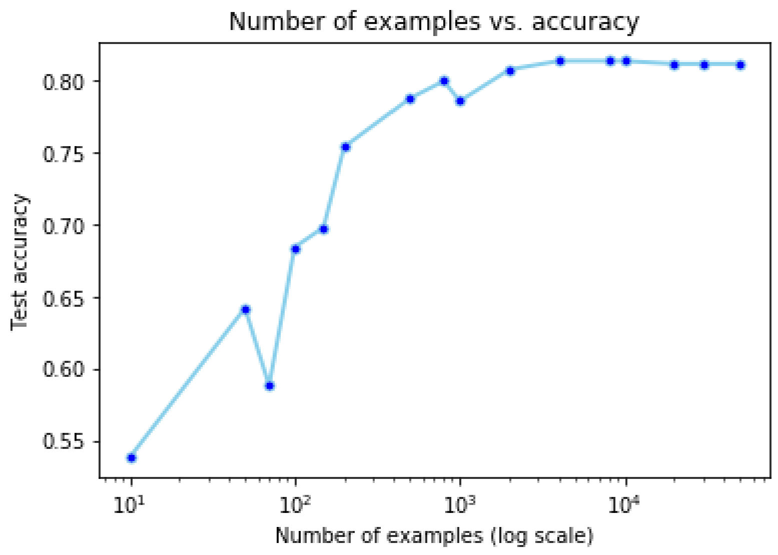
32 word2vec features, 500 examples test accuracy 0.752

64 word2vec features, 500 examples test accuracy 0.718

128 word2vec features, 500 examples test accuracy 0.738

```
import pylab
import matplotlib.pyplot as plt
a = [pow(10, i) for i in range(10)]
b = [pow(10, i*2) for i in range(9)]
b.append(12)

context = [1, 2, 3, 4, 5, 8, 16, 32, 64, 128]
```

```
accuracy = [0.766, 0.776, 0.762, 0.744, 0.774, 0.764, 0.746, 0.752, 0.718, 0.738]

examples = [10,50,70,100,150,200,500,800,1000,2000,4000,8000,10000, 20000, 30000,
accuracy = [0.538, 0.642, 0.588, 0.684, 0.698, 0.754, 0.788, 0.8, 0.786, 0.808, 0.

# different number of training examples:
# word = [0.496, 0.6, 0.584, 0.616, 0.678, 0.716, 0.76, 0.76, 0.784, 0.776, 0.784,
# lsa = [0.474, 0.578, 0.498, 0.614, 0.646, 0.686, 0.722, 0.74, 0.734, 0.75, 0.758
# combo = [0.496, 0.618, 0.614, 0.626, 0.682, 0.718, 0.766, 0.772, 0.784, 0.776, 0
# examples = [10,50,70,100,150,200,500,800,1000,2000,4000,8000,10000]

# word = [0.784, 0.784, 0.784, 0.784, 0.784, 0.784, 0.784, 0.784, 0.784, 0.784, 0.
# lsa = [0.476, 0.476, 0.604, 0.596, 0.69, 0.704, 0.712, 0.726, 0.728, 0.734, 0.73
# combo = [0.784, 0.784, 0.784, 0.778, 0.782, 0.784, 0.784, 0.784, 0.784, 0.784, 0
# examples = [1, 2, 5, 10, 50, 70, 100, 150, 200, 300, 500, 1000, 2000, 2999]
# context_size = [1,2,3,4,5,8,16,32,64,32,64,128]
# accuracy = [0.766,0.776, 0.762,0.774,0.744, 0.764, 0.746, 0.752, 0.718, 0.738]
# fig = plt.figure()
# ax = fig.add_subplot(2, 1, 1)

# plt.

# line, = ax.plot(a, color='blue', lw=2)
# line, = ax.plot(b, color = "green", lw =2)
# ax.set_yscale('log')
plt.plot( examples, accuracy, marker='o', markerfacecolor='blue', markersize=5, co
# plt.plot( examples, lsa, marker='o', markerfacecolor='darkred', markersize = 5,
# plt.plot( examples, combo, marker='o', markerfacecolor='forestgreen', markersize
plt.xscale("log")
plt.xlabel("Number of examples (log scale)")
plt.ylabel("Test accuracy")
plt.title("Number of examples vs. accuracy")
# plt.legend(loc = 'lower right', shadow=True,)

pylab.show()
```



```
**Part 2: Lab writeup**

Part 2 of your lab report should discuss any implementation details that were
important to filling out the code above. Then, use the code to set up
experiments that answer the following questions:

1. Qualitatively, what do you observe about nearest neighbors in representation
space? (E.g. what words are most similar to _the_, _dog_, _3_, and _good_?) How
well do word2vec representations correspond to your intuitions about word
similarity?

2. One important parameter in word2vec-style models is context size. How does
changing the context size affect the kinds of representations that are learned?

3. How do results on the downstream classification problem compare to
   part 1?

4. What are some advantages and disadvantages of learned embedding
representations, relative to the featurization done in part 1?

5. What are some potential problems with constructing a representation of the
review by averaging the embeddings of the individual words?
```

Part 2: Lab writeup

Part 2 of your lab report should discuss any implementation details that were important to filling out t
up experiments that answer the following questions:

1. Qualitatively, what do you observe about nearest neighbors in representation space? (E.g. what
   good?) How well do word2vec representations correspond to your intuitions about word similari
2. One important parameter in word2vec-style models is context size. How does changing the con
   representations that are learned?
3. How do results on the downstream classification problem compare to part 1?
4. What are some advantages and disadvantages of learned embedding representations, relative t
5. What are some potential problems with constructing a representation of the review by averagin

```
%%bash
!(stat -t /usr/local/lib/*/dist-packages/google/colab > /dev/null 2>&1) && exit
rm -rf 6864-hw1
git clone https://github.com/lingo-mit/6864-hw1.git
```

```
Cloning into '6864-hw1'...
```

```
import sys
sys.path.append("/content/6864-hw1")

import csv
import itertools as it
import numpy as np
np.random.seed(0)

import lab_util
```

Hidden Markov Models

In the remaining part of the lab (containing part 3) you'll use the Baum--Welch algorithm to learn *cate*
vocabulary. Answers to questions in this lab should go in the same report as the initial release.

As before, we'll start by loading up a dataset:

```
data = []
n_positive = 0
n_disp = 0
with open("/content/6864-hw1/reviews.csv") as reader:
  csvreader = csv.reader(reader)
  next(csvreader)
  for id, review, label in csvreader:
    label = int(label)

    # hacky class balancing
    if label == 1:
      if n_positive == 2000:
        continue
      n_positive += 1
    if len(data) == 4000:
      break

    data.append((review, label))

    if n_disp > 5:
      continue
    n_disp += 1
    print("review:", review)
    print("rating:", label, "(good)" if label == 1 else "(bad)")
    print()

print(f"Read {len(data)} total reviews.")
np.random.shuffle(data)
reviews, labels = zip(*data)
train_reviews = reviews[:3000]
train_labels = labels[:3000]
val_reviews = reviews[3000:3500]
val_labels = labels[3000:3500]
test_reviews = reviews[3500:]
test_labels = labels[3500:]
```

```
review: I have bought several of the Vitality canned dog food products and have 
rating: 1 (good)

review: Product arrived labeled as Jumbo Salted Peanuts...the peanuts were actua
rating: 0 (bad)

review: This is a confection that has been around a few centuries.  It is a ligh
rating: 1 (good)

review: If you are looking for the secret ingredient in Robitussin I believe I ha
rating: 0 (bad)

review: Great taffy at a great price.  There was a wide assortment of yummy taffy
rating: 1 (good)

review: I got a wild hair for taffy and ordered this five pound bag. The taffy wa
rating: 1 (good)

Read 4000 total reviews.
```

Next, implement the forward--backward algorithm for HMMs like we saw in class.

IMPORTANT NOTE: if you directly multiply probabilities as shown on the class slides, you'll get underf
the log domain (remember that `log(ab) = log(a) + log(b)`, `log(a+b) = logaddexp(a, b)`).

```
# hmm model
from scipy.special import logsumexp
from math import e

class HMM(object):
    def __init__(self, num_states, num_words):
        self.num_states = num_states
        self.num_words = num_words

        self.states = range(num_states)
        self.symbols = range(num_words)

        # initialize the matrix A with random transition probabilities p(j|i)
        # A should be a matrix of size `num_states x num_states`
        # A should be a matrix of size `num_states x num_states`
        # with rows that sum to 1
        temp_A = np.random.rand(self.num_states, self.num_states)
        self.A = (temp_A.T/temp_A.sum(axis=1)).T

        # initialize the matrix B with random emission probabilities p(o|i)
        # B should be a matrix of size `num_states x num_words`
        # with rows that sum to 1
        ## B[i][o] = p(o|i)
        temp_B = np.random.rand(self.num_states, self.num_words)
        self.B = (temp_B.T/temp_B.sum(axis=1)).T

        # initialize the vector pi with a random starting distribution
        # pi should be a vector of size `num_states`
        temp_pi = np.random.rand(self.num_states)
        self.pi = temp_pi/sum(temp_pi)

    def generate(self, n):
        """randomly sample the HMM to generate a sequence.
        """
        # we'll give you this one

        sequence = []
        # initialize the first state
        state = np.random.choice(self.states, p=self.pi)
        for i in range(n):
            # get the emission probs for this state
            b = self.B[state, :]
            # emit a word
            ######## exponentiate b here to bring out of log space ####
            word = np.random.choice(self.symbols, p=b)
            sequence.append(word)
            # get the transition probs for this state
            a = self.A[state, :]
            # update the state
            state = np.random.choice(self.states, p=a)
        return sequence

    def forward(self, obs):
        # run the forward algorithm
        # this function should return a `len(obs) x num_states` matrix
        # where the (i, j)th entry contains p(obs[:t], hidden_state_t = i)
        # where the (i, j)th entry contains p(obs[:t], hidden_state_t = j)
        alpha = np.zeros((len(obs), self.num_states))
        # alpha = np.full((len(obs), self.num_states), 0.00000000000000000001)
        for t in range(len(alpha)):
            for j in range(len(alpha[0])):
                if t==0:
                    alpha[t][j]=np.log(self.pi[j])+np.log(self.B[j][obs[t]])
                else:
                    alpha_t_1 = alpha[t-1]
                    aij = np.log(self.A[:,j])
                    alpha[t][j] = logsumexp(alpha_t_1 + aij)+np.log(self.B[j][obs[

        # def find(t, j):
        #     # if alpha[t][j] != 0:
        #     #     return alpha[t][j]
        #     if t == 0:
        #       alpha[t][j] = np.log(self.pi[j])+np.log(self.B[j][obs[t]])
        #     else:
        #       total = None
        #       for i in range(self.num_states):
        #           # print("yoyoyo")
        #           if total == None:
        #               total = find(t-1, i) + np.log(self.A[i][j])
        #           else:
        #               total = np.logaddexp(total, find(t-1, i)+np.log(self.A[i][
        #       total += np.log(self.B[j][obs[t]])
        #       alpha[t][j] = total
        #     return alpha[t][j]

        # log_probs = None
        # # print("number of states", (self.num_states))
        # for state in range(self.num_states):
        #     if log_probs == None:
        #         # log_probs = find(len(obs)-1, state)
        #         log_probs = alpha[len(obs)-1, state]
        #     else:
        #         # log_probs = np.logaddexp(log_probs, find(len(obs)-1, state))
        #         log_probs = np.logaddexp(log_probs, alpha[len(obs)-1, state])
        # print("this ur log prob", logsumexp(alpha, axis = 1)[-1])
        # print("this ur log prob part 2 kkokokoko", log_probs)
        return alpha, logsumexp(alpha, axis = 1)[-1]

    def backward(self, obs):
        # run the backward algorithm
        # this function should return a `len(obs) x num_states` matrix
        # where the (i, j)th entry contains p(obs[t+1:] | hidden_state_t = i)
        # where the (i, j)th entry contains p(obs[t+1:] | hidden_state_t = j)

        beta = np.zeros((len(obs), self.num_states))
        for t in range(len(beta)-1, -1, -1):
            for i in range(len(beta[0])):
                if t == len(obs)-1:
                    beta[t][i] = np.log(1)
                else:
                    beta_tplus1 = beta[t+1]
                    b_j_o_tplus1 = np.log(self.B[:,obs[t+1]])
                    aij = np.log(self.A[i])
                    beta[t][i] = logsumexp(beta_tplus1 + b_j_o_tplus1 + aij)

        # def find(t, i):
        #     if beta[t][i] != 0:
        #       return beta[t][i]
        #     if t == len(obs)-1:
        #       beta[t][i] = np.log(1)
        #     else:
        #       total = 0
        #       for j in range(self.num_states):
        #         # total += log(self.A[i][j])*log(self.B[j][obs[t+1]])*find(t+1,
        #           # total = np.logaddexp(total, np.log(self.A[i][j])+np.log(self
        #           total = np.logaddexp(total, np.log(self.A[i][j])+np.log(self.B
        #       beta[t][i] = total
        #     return beta[t][i]

        # log_probs = None

        # for state in range(self.num_states):
        #     if log_probs == None:
        #         log_probs = np.log(self.pi[state])+np.log(self.B[state][obs[0]])
        #     else:
        #         log_probs = np.logaddexp(log_probs, np.log(self.pi[state])+np.lo
        # print("log probs forward", log_probs)
        # print(logsumexp(alpga, axis = 0))
        log_probs = logsumexp(np.log(self.pi)+np.log(self.B[:,obs[0]])+beta[0])
        return beta, log_probs

    def forward_backward(self, obs):
        # compute forward--backward scores

        # logprob is the total log-probability of the sequence obs
        # (marginalizing over hidden states)

        # gamma is a matrix of size `len(obs) x num_states`
        # it contains the marginal probability of being in state i at time t

        # xi is a tensor of size `len(obs) x num_states x num_states`
        # it conains the marginal probability of transitioning from i to j at t

        # fix log prob stuff for these
        alpha, log_probs1 = self.forward(obs)
        beta, log_probs = self.backward(obs)
        # print("log probs for forward", "log probs for backward")
        # print(log_probs1, log_probs)
        # logprob = math.log(probs1, 10)
        # gamma = 1/probs_1*np.multiply(alpha,beta)
        gamma = alpha+beta-log_probs

        xi = np.zeros((len(obs)-1, self.num_states, self.num_states))
        for t in range(len(xi)):
            for i in range(len(xi[0])):
                for j in range(len(xi[0][0])):
                    xi[t][i][j] = alpha[t][i]+np.log(self.A[i][j])+np.log(self.B[j

        # return logprob, xi, gamma
        return log_probs, xi, gamma

    def update_expected_sij(self, expected_sij, xi):
        expected_sij = logsumexp(xi, axis = 0)
        # for i in range(len(expected_sij)):
        #     for j in range(len(expected_sij[0])):
        #         total = 0
        #         for t in range(len(xi)):
        #             total=np.logaddexp(xi[t][i][j], total)
        #         expected_sij[i][j] = np.logaddexp(expected_sij[i][j], total)
        return expected_sij

    def update_expected_si(self, expected_si, gamma):
        # print(type(expected_si), type(gamma[:-1].sum(axis=0)))
        # expected_si += gamma[:-1].sum(axis = 0)
        expected_si = np.logaddexp(expected_si, logsumexp(gamma[:-1], axis = 0))
        return expected_si

    def update_expected_sj_wk(self, expected_sj_wk, gamma, obs):
        for j in range(len(expected_sj_wk)):
            for k in range(len(expected_sj_wk[0])):
            # for k in range(len(expected_sj_wk[0])):
                total = 0
                for t in range(len(gamma)):
                    if obs[t] == k:
                        # total.append(gamma[t][j])
                        total = np.logaddexp(total, gamma[t][j])
                expected_sj_wk[j][k] = np.logaddexp(expected_sj_wk[j][k], logsumex
        return expected_sj_wk

    def update_expected_sj(self, expected_sj, gamma):
        # expected_sj += gamma.sum(axis = 0)
        expected_sj = np.logaddexp(expected_sj, logsumexp(gamma, axis = 0))
        return expected_sj

    def update_expected_pi_si(self, expected_pi_si, gamma):
        # for s in range(len(expected_pi_si)):
        #     expected_pi_si[s] = np.logaddexp(gamma[0][s], expected_pi_si[s])
        expected_pi_si = np.logaddexp(expected_pi_si, gamma[0])
        return expected_pi_si

    def learn_unsupervised(self, corpus, num_iters):
        """Run the Baum Welch EM algorithm
        """

        for i_iter in range(num_iters):
            expected_si = np.zeros(self.num_states)
            expected_sij = np.zeros((self.num_states, self.num_states))
            expected_sj_wk = np.zeros((self.num_states, self.num_words))
            expected_sj = np.zeros(self.num_states)
            expected_pi_si = np.zeros(self.num_states)
            # keep running sum of these expected values and update at the end
            total_logprob = 0
            # for review in corpus:
            for i in range(len(corpus)):
                logprobs, xi, gamma = self.forward_backward(corpus[i])
                expected_si = self.update_expected_si(expected_si, gamma)
                expected_sij = self.update_expected_sij(expected_sij, xi)
                expected_sj_wk = self.update_expected_sj_wk(expected_sj_wk, gamma,
                expected_sj = self.update_expected_sj(expected_sj, gamma)
                expected_pi_si = self.update_expected_pi_si(expected_pi_si, gamma)

                total_logprob += logprobs
                print("review number " + str(i/len(corpus)), "running avg logprobs
                # your code here
            total_logprob/=len(corpus)
            # expected_pi_si -= np.log(len(corpus))
            print("log-likelihood", total_logprob)
            self.A = np.exp(expected_sij - np.array([expected_si]).T)
            self.B = np.exp(expected_sj_wk - np.array([expected_sj]).T)
            self.pi = np.exp(expected_pi_si - np.log(len(corpus)))


##### when calculating xiand gamma, instead of dividing, do log space
```

```
import sys
sys.setrecursionlimit(100000)

tokenizer = lab_util.Tokenizer()
tokenizer.fit(train_reviews)
train_reviews_tk = tokenizer.tokenize(train_reviews)
print(tokenizer.vocab_size)

hmm = HMM(num_states=2, num_words=tokenizer.vocab_size)
hmm.learn_unsupervised(train_reviews_tk, 3)
```

```
review number 0.962 running avg logprobs -1062.157397154726
review number 0.9623333333333334 running avg logprobs -1061.8946071832434
review number 0.9626666666666667 running avg logprobs -1061.926349016642
review number 0.963 running avg logprobs -1063.1042207010842
review number 0.9633333333333334 running avg logprobs -1063.2674157437941
review number 0.9636666666666667 running avg logprobs -1063.3681409559642
review number 0.964 running avg logprobs -1063.5672450028985
review number 0.9643333333333334 running avg logprobs -1063.320304574298
review number 0.9646666666666667 running avg logprobs -1064.1019187877575
review number 0.965 running avg logprobs -1064.0663782868178
review number 0.9653333333333334 running avg logprobs -1063.8245216220428
review number 0.9656666666666667 running avg logprobs -1063.6150816096842
review number 0.966 running avg logprobs -1063.6939759643003
review number 0.9663333333333334 running avg logprobs -1063.5672332877375
review number 0.9666666666666667 running avg logprobs -1063.7297829569695
review number 0.967 running avg logprobs -1064.1509413351341
review number 0.9673333333333334 running avg logprobs -1064.0797816722747
review number 0.9676666666666667 running avg logprobs -1064.0972471765526
review number 0.968 running avg logprobs -1063.9799028016705
review number 0.9683333333333334 running avg logprobs -1064.124254247945
review number 0.9686666666666667 running avg logprobs -1063.8124026548999
review number 0.969 running avg logprobs -1063.9633028429332
review number 0.9693333333333334 running avg logprobs -1063.8266494247744
review number 0.9696666666666667 running avg logprobs -1063.697986806636
review number 0.97 running avg logprobs -1063.7893104420193
review number 0.9703333333333334 running avg logprobs -1063.9848607859055
review number 0.9706666666666667 running avg logprobs -1063.765397467895
review number 0.971 running avg logprobs -1064.0392052736343
review number 0.9713333333333334 running avg logprobs -1063.8472830168068
review number 0.9716666666666667 running avg logprobs -1063.6840365522569
review number 0.972 running avg logprobs -1063.5906521659522
review number 0.9723333333333334 running avg logprobs -1063.3144391070325
review number 0.9726666666666667 running avg logprobs -1063.0619131026444
review number 0.973 running avg logprobs -1063.0020595609633
review number 0.9733333333333334 running avg logprobs -1062.9212060327463
review number 0.9736666666666667 running avg logprobs -1062.7174510063858
review number 0.974 running avg logprobs -1063.6005840632545
review number 0.9743333333333334 running avg logprobs -1064.3341276631725
review number 0.9746666666666667 running avg logprobs -1064.3563951891138
review number 0.975 running avg logprobs -1064.1116946352095
review number 0.9753333333333334 running avg logprobs -1063.8496161527394
review number 0.9756666666666667 running avg logprobs -1065.3548202335173
review number 0.976 running avg logprobs -1065.7842959140053
review number 0.9763333333333334 running avg logprobs -1066.902862799503
review number 0.9766666666666667 running avg logprobs -1066.670443580452
review number 0.977 running avg logprobs -1066.5200145029307
review number 0.9773333333333334 running avg logprobs -1066.607726168395
review number 0.9776666666666667 running avg logprobs -1066.6084921284385
review number 0.978 running avg logprobs -1067.1130946286205
review number 0.9783333333333334 running avg logprobs -1066.8482902793787
review number 0.9786666666666667 running avg logprobs -1066.6738124658607
review number 0.979 running avg logprobs -1068.422659255624
review number 0.9793333333333333 running avg logprobs -1068.3662495807882
review number 0.9796666666666667 running avg logprobs -1068.6801265062286
review number 0.98 running avg logprobs -1069.3116851430218
review number 0.9803333333333333 running avg logprobs -1069.994828485061
review number 0.9806666666666667 running avg logprobs -1069.8750150969433
review number 0.981 running avg logprobs -1070.3821626660251
review number 0.9813333333333333 running avg logprobs -1070.1616712763657
review number 0.9816666666666667 running avg logprobs -1070.1171897253123
review number 0.982 running avg logprobs -1069.8549228822762
review number 0.9823333333333333 running avg logprobs -1070.1956668534388
review number 0.9826666666666667 running avg logprobs -1070.088952966992
review number 0.983 running avg logprobs -1069.876348020101
review number 0.9833333333333333 running avg logprobs -1069.7057804202936
review number 0.9836666666666667 running avg logprobs -1069.6241006777782
review number 0.984 running avg logprobs -1069.9550459285465
review number 0.9843333333333333 running avg logprobs -1069.8123047827496
review number 0.9846666666666667 running avg logprobs -1070.0416584564175
review number 0.985 running avg logprobs -1069.923133429742
review number 0.9853333333333333 running avg logprobs -1069.7584335778188
review number 0.9856666666666667 running avg logprobs -1070.0008496186256
review number 0.986 running avg logprobs -1069.8059744752754
review number 0.9863333333333333 running avg logprobs -1069.628770334747
review number 0.9866666666666667 running avg logprobs -1069.453749418545
review number 0.987 running avg logprobs -1069.570837669485
review number 0.9873333333333333 running avg logprobs -1069.402906208953
review number 0.9876666666666667 running avg logprobs -1069.6182495907176
review number 0.988 running avg logprobs -1070.4922112713912
review number 0.9883333333333333 running avg logprobs -1070.607976015914
review number 0.9886666666666667 running avg logprobs -1074.4468006899565
review number 0.989 running avg logprobs -1074.2623902009475
review number 0.9893333333333333 running avg logprobs -1074.0831938113952
review number 0.9896666666666667 running avg logprobs -1073.8482979800897
review number 0.99 running avg logprobs -1073.696307886683
review number 0.9903333333333333 running avg logprobs -1073.4525839590042
review number 0.9906666666666667 running avg logprobs -1073.324343127259
review number 0.991 running avg logprobs -1073.5337005580682
review number 0.9913333333333333 running avg logprobs -1073.544179302291
review number 0.9916666666666667 running avg logprobs -1073.3940775941655
review number 0.992 running avg logprobs -1073.5936617398886
review number 0.9923333333333333 running avg logprobs -1073.4594806432763
review number 0.9926666666666667 running avg logprobs -1073.318911957182
review number 0.993 running avg logprobs -1073.0636374281155
review number 0.9933333333333333 running avg logprobs -1072.8088089945118
review number 0.9936666666666667 running avg logprobs -1072.6098841559703
review number 0.994 running avg logprobs -1072.9425897524973
review number 0.9943333333333333 running avg logprobs -1073.472007249375
review number 0.9946666666666667 running avg logprobs -1073.3573730914334
review number 0.995 running avg logprobs -1073.5447796930807
review number 0.9953333333333333 running avg logprobs -1073.4817323888103
review number 0.9956666666666667 running avg logprobs -1073.8283746158627
review number 0.996 running avg logprobs -1074.138545088336
review number 0.9963333333333333 running avg logprobs -1074.2437376237583
review number 0.9966666666666667 running avg logprobs -1074.1739200197476
review number 0.997 running avg logprobs -1074.1217170253085
review number 0.9973333333333333 running avg logprobs -1073.8631953663871
review number 0.9976666666666667 running avg logprobs -1073.7332319985883
review number 0.998 running avg logprobs -1073.4920253646653
review number 0.9983333333333333 running avg logprobs -1073.530135631122
review number 0.9986666666666667 running avg logprobs -1073.3533881816213
review number 0.999 running avg logprobs -1073.0970880457858
review number 0.9993333333333333 running avg logprobs -1073.0767006003568
review number 0.9996666666666667 running avg logprobs -1072.9330995010323
log-likelihood -1072.575455134532
```