6.864 Adv. Natural Language Processing Homework 1

Anonymous Author

1 Word Embeddings

In this section, we investigated the ability of multiple word embedding methods to encode the meaning of words based on learning from a text corpus. In particular, we focused on two methods: latent semantic analysis (LSA) and the Word2Vec model. In addition to evaluating embeddings by qualitatively observing nearest neighbors for select words, we also evaluated them on a down-stream logistic regression modeling problem. For both the unsupervised learning of word embeddings and model training, we used a dataset of 3500 labelled reviews with a vocabulary size of 2006, each of which had a $0-1$ label indicating if the review was bad or good, respectively. We set aside 500 reviews for the test dataset, making the remainder available for training.

1.1 Implementation Details

1.1.1 Latent Semantic Analysis

The first word embedding method we implemented was LSA. In LSA, we encode our text corpus as a term-document matrix W_{td}, where the entry w_{td} is the number of times term t appears in document d. This matrix is normalized with term frequency-inverse document frequency (TF-IDF) normalizationWe then factorize this matrix using singular-value decomposition (SVD), getting a decomposition of the form $U\Sigma V^T$, where U, V are unitary and Σ is diagonal. We truncate U to a pre-determined embedding size to get our word embeddings, which we can then use to encode words in downstream tasks.

To implement the matrix factorization, we used the `svd` method in the `numpy.linalg` library to get the matrix U, which we then truncated the rows to the appropriate embedding size. We had considered the use of `TruncatedSVD` from scikit-learn, but found it inappropriate for our use case since it returns $U\Sigma$ instead of U. In addition, to implement TF-IDF normalization, we used numpy to calculate the normalization factor for each term.

1.1.2 Word2Vec

For the Word2Vec embeddings, we used the continuous bag-of-words (CBOW) formulation of the problem. We implemented a feed-forward neural network with a single hidden-layer of the embedding dimension size in PyTorch. Like in the original Word2Vec paper, we did not incorporate a non-linear activation function. This network takes in the context words around the word we want to predict, converts them into their learned embeddings in the hidden layer, and passes their average through the second weight matrix and a softmax activation function to get the prediction of the hidden word. For numeric stability, we used the log-softmax instead of softmax. This model was then trained on the reviews text corpus to get the learned word embeddings.

1.2 Questions

1.2.1 Latent Semantic Analysis

Question 1 and 2 After performing LSA with a embedding size of 500, we see that the nearest neighbors generally relate in meaning (see Table 1), either as synonym, antonyms, or topically, to the given word. For example, “bad" had nearest neighbors such as “disgusting" and “awful," which are similar in meaning, as well as words with opposite meanings like “positive." We see that this holds true for various embedding dimensions. For embedding sizes of similar magnitudes, like 100, we observed similar qualitative performance, with the embeddings having better, more related results for cookie like “cookies," “oreos," and “craving," but worse results for “bad." As we moved further from these sizes, the nearest neighbors tended to become qualitatively worse, with fewer similar words..

Question 3 Instead of the term-document matrix, we could have used the term-term matrix

Embed Size	bad	cookie	4
10	agree, entirely, positive, forward, overly	cookies, muffins, cake, tough, excellent	1, 6, 5, protein, 7
100	taste, strange, like, myself, nasty	cookies, nana's, oreos, bars, craving	1, 6, 70, concentrated, measure
500	disgusting, awful, positive, bland, gone	nana's, moist, odd, impossible, needs	mistake, 2nd, toast, table, 70
1000	disgusting, touch, wild, entirely, timely	nana's, moist, needs, chewy, odd	economical, mistake, total, 70, certainly

Table 1: Nearest neighbors for select words at various LSA embedding sizes. Words are order from nearest to farthest. We bolded the default size given in the lab. See code for more results.

$W_{tt} = W_{td}W_{td}^T$. Since V in the SVD decomposition $U\Sigma V^T$ is unitary, we know that $V^TV = I$ and get that

$$W_{tt} = W_{td}W_{td}^T = U\Sigma V^T V \Sigma^T U^T = USU^T$$

where S is diagonal. Thus we'd expect that the SVD decomposition of W_{tt} would result in the same word embeddings as W_{td}. In practice however, while the most columns do appear the same up to a constant factor of -1, the final columns are different. This could possibly be because of numeric instability and underflows, as the values tend to be very low in these columns.

Question 4 We found that in general, the LSA embeddings improved model performance, as seen in Figure 1. This effect was observed over various training data sizes, indicating the word embeddings' consistency in improving performance.

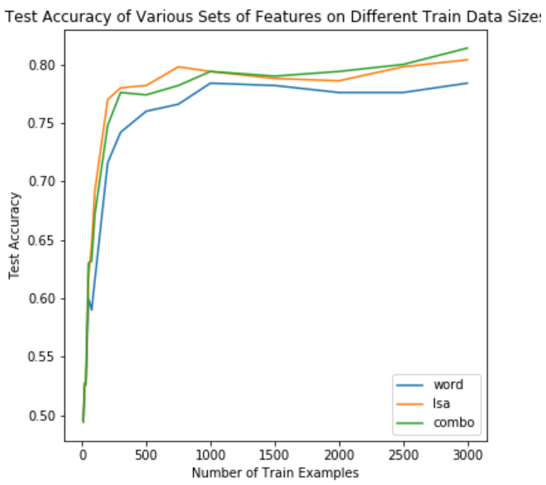


Figure 1: Test accuracy of various sets of features from LSA

Question 5 We found that embedding size is a hyperparameter in need of tuning. Test accuracy would increase with embedding size until around an embedding size of 500. At this point, test accuracy plateaued or even decreases at some train data sizes. The results are depicted in Figure 2.

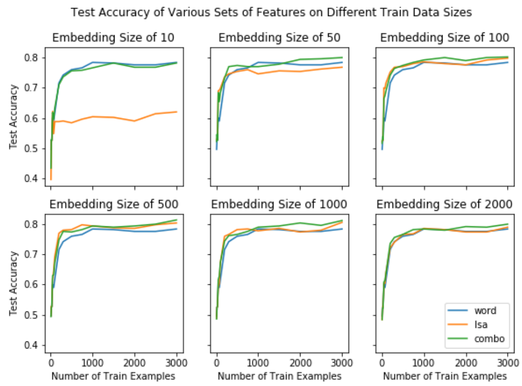


Figure 2: Test accuracy of various sets of features from LSA, over different embedding sizes

1.2.2 Word2Vec

Question 1 Qualitatively, we saw that the nearest neighbors to selected words did not seem to share a close similarity in meaning (see Table 2). Instead, adjectives like "bad" seemed to have many nouns, which makes sense as these are nouns that likely are described as bad and thus would have the word near them. Similarly, the word "cookie" had many nouns around it as well, which likely are also found close "cookie" in the corpus. This differed from our intuitive understanding of word similarity to be about meaning, which was better reflected in LSA.

Question 2 If we vary the context size in Word2Vec, we see still get similar types of nearest neighbors as those described in Question 1 (see Table 2). These results seem to worsen if we decrease the context size to the minimal 1, and improve if we increase the context size. This seems reasonable, as larger context sizes allow us to see more of the local neighbor of each word, and therefore notice related words that appear close but not right next to the word (i.e. within 5-10 words, but not 2).

Question 3 For the default hyperparameters and an embedding size of 500, as shown in Figure 3, we see that LSA outperforms Word2Vec on the test dataset in almost all train data sizes. The exception is a size of 2000; however, even in this case, LSA and Word2Vec are very close in test accuracy. We then varied the embedding size to get the results shown in Figure 4. While LSA does outperform Word2Vec in smaller embedding sizes, Word2Vec outperforms LSA at larger sizes such as 1000 and 2000. In addition, we note that like LSA, Word2Vec test accuracy increases with embedding sizes, although it plateaus later at the sizes of 1000 and 2000. This indicates that Word2Vec requires more dimensions to encode the same amount of the word similarity, but can ultimately provide better performance.

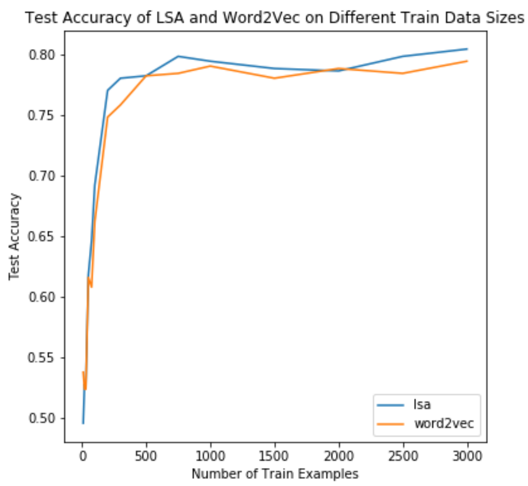


Figure 3: Test accuracy of LSA and Word2Vec over various train data sizes

Question 4 As we observed in figures for question 3 (Figures 3 and 4), while Word2Vec can achieve an overall better test accuracy than LSA at large enough embedding sizes, LSA is more space-efficient, as it can achieve the same test accuracy as Word2Vec at a lower embedding size until it begins plateauing. In addition, our qualitative analysis of nearest neighbors indicates that LSA

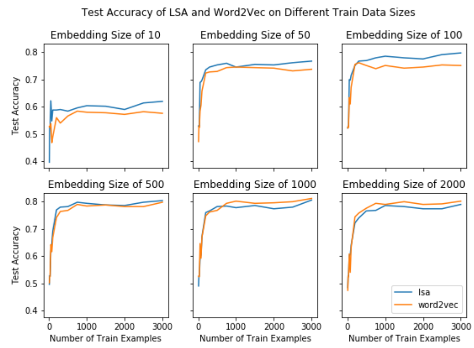


Figure 4: Test accuracy of LSA and Word2Vec over various train data sizes and embedding sizes

and Word2Vec tend to encode different types of word similarity. In particular, LSA seems more focused on meaning, presenting many synonyms and antonyms, whereas Word2Vec seems more focused on word proximity in sentences.

Question 5 One design choice in our implementation of the Word2Vec model was to average the embeddings of the context words. This can be problematic, as if two words have embeddings with opposite values of similar magnitude, the resulting vector will lose the signed information and pass along a zero to the output layer. If the magnitude of certain embedding dimensions actually encodes some meaningful information, say positivity vs. negativity, the intensity of this information can be lost when combining opposite vectors. This could be undesirable, as the unsigned intensity of this information might affect the probability of various words and would be lost in the averaging schema.

2 Hidden Markov Models

2.1 Implementation Details

To implement our hidden Markov model (HMM), we used Pytorch and the formulation introduced in lecture, training it with a expectation-maximization (EM) algorithm. We implemented the forward-backward algorithm, and used it along with the Baum-Welch EM algorithm to train our HMM. We used the same review corpus as introduced in Section 1. As we would work with discrete probability distributions with many states, we performed our calculations in log-space for numeric stability and preventing underflows in marginal probabilities. All models were trained for 10 epoches given the time and resource constraints of the experiment, unless otherwise specified.

Context	bad	cookie
1	cons, messy, inches, pound, pleasant	personally, highly, boy, support, cholesterol
2	inches, done, banana, wanting, weak	he's, caused, substitute, bit, common
5	17, strange, trust, overpowering, salty	carrot, supermarket, family, death, disappointed
10	sodas, clearly, cooking, stomach, thin	vanilla, thinking, substitute, craving, sorry

Table 2: Nearest neighbors for select words at various Word2Vec context sizes. Words are order from nearest to farthest. We bolded the default size given in the lab. See code for more results.

2.2 Questions

Question 1 We list select clusters for the HMM at various numbers of states in Table 4. When there are only two states, these clusters seem fairly meaningless, as the two states share many of the same most probably words and these words tend to be common words, like "a," "the," and "is," or punctuation marks. When we increases the number of states to 10 or even 50, we see a similar problem with the most common English words and punctuation marks appearing in multiple states. However, we do begin to see some longer words, such as "have" and "these" that previously did not appear. However, if we generate sentences, we see that for 10 or 50 states, less common words are selected and more coherent text samples are returned, although the results are not close to human writing.

Unfortunately, due to the performance of our HMM implementation, we could not run the experiment for 100 states.

S	**Example State Clusters**
2	. a it and i , is br to <unk>
	<unk> the , i of . to and in that
10	. i <unk> it , they that and not you
	i a , have and the is you . !
50	<unk> . , and to it of for have not
	, . the to my is this a are these

Table 3: Select state clusters for trained with $S = 2, 10,$ and 50 states, presenting the ten most probable words of each cluster. Words are order from most probable to least. Note that punctuation count as words for the purposes of this experiment. See code for full results.

Question 2 We found that the logistic regression model fails to converge, even with 50 states and all 3000 available training points. We observe a spike in test accuracy around 500 training examples, after which accuracy slowly increases. For sizable training datasets, test accuracy lies between 0.6 and 0.65, much lower than test accuracy for either methods in Section 1, which had peak accuracies around 0.81.

This may in part be due to too small of an HMM, as well as the fact that an EM algorithm is only guaranteed to converge to a point of zero gradient, which might not be locally optimal much less globally so. Altogether, these concerns may provide an explanation to why our HMM models do not provide coherent sentences. In terms of the classification task, the word embeddings may be more appropriate for this task, as they encode some sense of word similarity that could include factors that strongly correlate to sentiment, whereas the distribution of next word from the HMM seems less tailored for such a task.

S	**Generated Text Samples**
2	'are not <unk> are are not not are <unk> these'
10	'during <unk> i up all , br i in single'
	'snack salt changed saw not , ! from have spent'
50	'allergic a it br over you purchase only about for'
	'just . this still if i are eaten the i'

Table 4: Select text samples of length 10 generated by HMMs trained with $S = 2, 10,$ and 50 states. See code for full results.

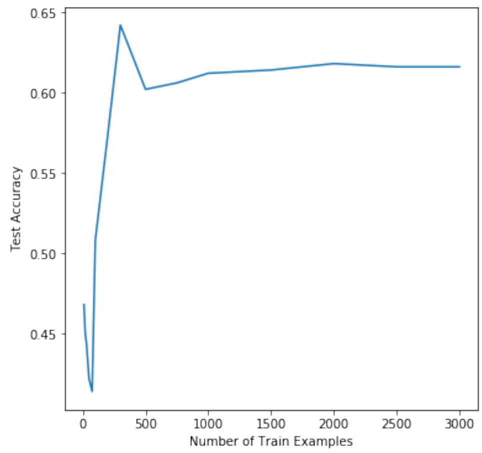


Figure 5: Test accuracy of the HMM with 50 states on various train data sizes

6864_hw1

March 1, 2020

In [1]:
```
%%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'...
```

In [0]:
```
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
```

In [0]:
```
from matplotlib import pyplot as plt
from google.colab import files
```

0.1 Introduction

In this lab, you'll explore three different ways of using unlabeled text data to learn pretrained word representations. Your lab report will describe the effects of different modeling decisions (representation learning objective, context size, etc.) on both qualitative properties of learned 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 place to upload them before the due date, but are still sorting out some logistics.) Please upload PDFs, with a maximum of four single-spaced pages. (If you want you can use the Association for Computational Linguistics style files.) Reports should have one section for each part of the homework assignment below. Each section should describe the details of your code implementation, and include whatever charts / tables are necessary to answer the set of 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:

In [4]:
```
data = []
n_positive = 0
```

```python
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 found them all
rating: 1 (good)

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

review: This is a confection that has been around a few centuries.  It is a light, pillowy citr
rating: 1 (good)

review: If you are looking for the secret ingredient in Robitussin I believe I have found it.
rating: 0 (bad)

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

```
review: I got a wild hair for taffy and ordered this five pound bag. The taffy was all very en
rating: 1 (good)

Read 4000 total reviews.
```

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

0.2 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 file browser on the left if you want to see how this works).

```
In [5]: 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]}")
```

```
TD matrix is 2006 x 3000
```

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

```
In [0]: from numpy.linalg import svd

        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.

          u, _, _ = svd(matrix)
          return u[:, :rep_size]
```

Let's look at some representations:

```
In [7]: reps = learn_reps_lsa(td_matrix, 500)
        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)
```

```
good 47
  gerber 1.873
  luck 1.885
  crazy 1.890
  flaxseed 1.906
  suspect 1.907
```

```
bad 201
  disgusting 1.625
  horrible 1.776
  shortbread 1.778
  gone 1.778
  dont 1.802
cookie 504
  nana's 0.964
  bars 1.363
  odd 1.402
  impossible 1.459
  cookies 1.484
jelly 351
  twist 1.099
  cardboard 1.197
  peanuts 1.311
  advertised 1.331
  plastic 1.510
dog 925
  happier 1.670
  earlier 1.681
  eats 1.702
  stays 1.722
  standard 1.727
the 36
  suspect 1.953
  flowers 1.961
  leaked 1.966
  m 1.966
  burn 1.967
4 292
  shortbread 1.674
  toast 1.683
  mistake 1.690
  2nd 1.701
  icing 1.723
```

We've been operating on the raw count matrix, but in class we discussed several reweighting schemes aimed at making LSA representations more informative.

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

```
In [0]: def transform_tfidf(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.
```

```
    num_docs = matrix.shape[1]
    tf_matrix = matrix.copy()
    td_occurrence = np.sum(np.where(tf_matrix > 0, 1, 0), axis=1, keepdims=True)
    idf = np.log(num_docs / td_occurrence)
    return tf_matrix * idf
```

How does this change the learned similarity function?

```
In [9]: td_matrix_tfidf = transform_tfidf(td_matrix)
        reps_tfidf = learn_reps_lsa(td_matrix_tfidf, 500)
        lab_util.show_similar_words(vectorizer.tokenizer, reps_tfidf, show_tokens)
```

```
good 47
  crazy 1.695
  gerber 1.753
  beat 1.758
  homemade 1.785
  tasting 1.799
bad 201
  disgusting 1.623
  awful 1.713
  positive 1.715
  bland 1.731
  gone 1.736
cookie 504
  nana's 1.103
  moist 1.388
  odd 1.452
  impossible 1.486
  needs 1.509
jelly 351
  twist 1.156
  cardboard 1.211
  advertised 1.402
  plum 1.447
  sold 1.470
dog 925
  happier 1.641
  earlier 1.658
  foods 1.690
  stays 1.697
  eats 1.704
the 36
  <unk> 1.478
  and 1.578
  . 1.581
  of 1.627
  is 1.632
```

```
4 292
  mistake 1.687
  2nd 1.707
  toast 1.708
  table 1.714
  70 1.723
```

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 embeddings.

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

```
In [10]: 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 = xs@reps_tfidf

           # 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, verbose=True):
           xs_featurized = featurizer(xs)
           pred_ys = model.predict(xs_featurized)
           acc = np.mean(pred_ys == ys)
           if verbose: print("test accuracy", acc)
           return acc

         def training_experiment(name, featurizer, n_train, verbose=True):
           if verbose: 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)
    acc = eval_model(model, featurizer, test_xs, test_ys, verbose=verbose)
    if verbose: print()
    return acc

training_experiment("word", word_featurizer, 10)
training_experiment("lsa", lsa_featurizer, 10)
training_experiment("combo", combo_featurizer, 10)
print()
```

```
word features, 10 examples
test accuracy 0.496

lsa features, 10 examples
test accuracy 0.496

combo features, 10 examples
test accuracy 0.494
```

Part 1: Lab writeup

Part 1 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*?)
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 prove about the relationship between the left singular vectors of W_{td} and W_{tt}? Do you observe this behavior with your implementation of `learn_reps_lsa`? Why or why not?
4. Do learned representations help with the review classification problem? What is the relationship between the number of labeled examples and the effect of word embeddings?
5. What is the relationship between the size of the word embeddings and their usefulness for the classification task.

```
In [11]: train_sizes = [10, 20, 30, 50, 75, 100, 200, 300, 500, 750, 1000, 1500, 2000, 2500, 30

         word_500_results = []
```

```python
lsa_500_results = []
combo_500_results = []

reps_tfidf = learn_reps_lsa(td_matrix_tfidf, 500)

for n in train_sizes:
    word_500_results.append(training_experiment("word", word_featurizer, n))
    lsa_500_results.append(training_experiment("lsa", lsa_featurizer, n))
    combo_500_results.append(training_experiment("combo", combo_featurizer, n))
```

```
word features, 10 examples
test accuracy 0.496

lsa features, 10 examples
test accuracy 0.496

combo features, 10 examples
test accuracy 0.494

word features, 20 examples
test accuracy 0.526

lsa features, 20 examples
test accuracy 0.528

combo features, 20 examples
test accuracy 0.524

word features, 30 examples
test accuracy 0.526

lsa features, 30 examples
test accuracy 0.526

combo features, 30 examples
test accuracy 0.528

word features, 50 examples
test accuracy 0.6

lsa features, 50 examples
test accuracy 0.618

combo features, 50 examples
test accuracy 0.63

word features, 75 examples
test accuracy 0.59
```

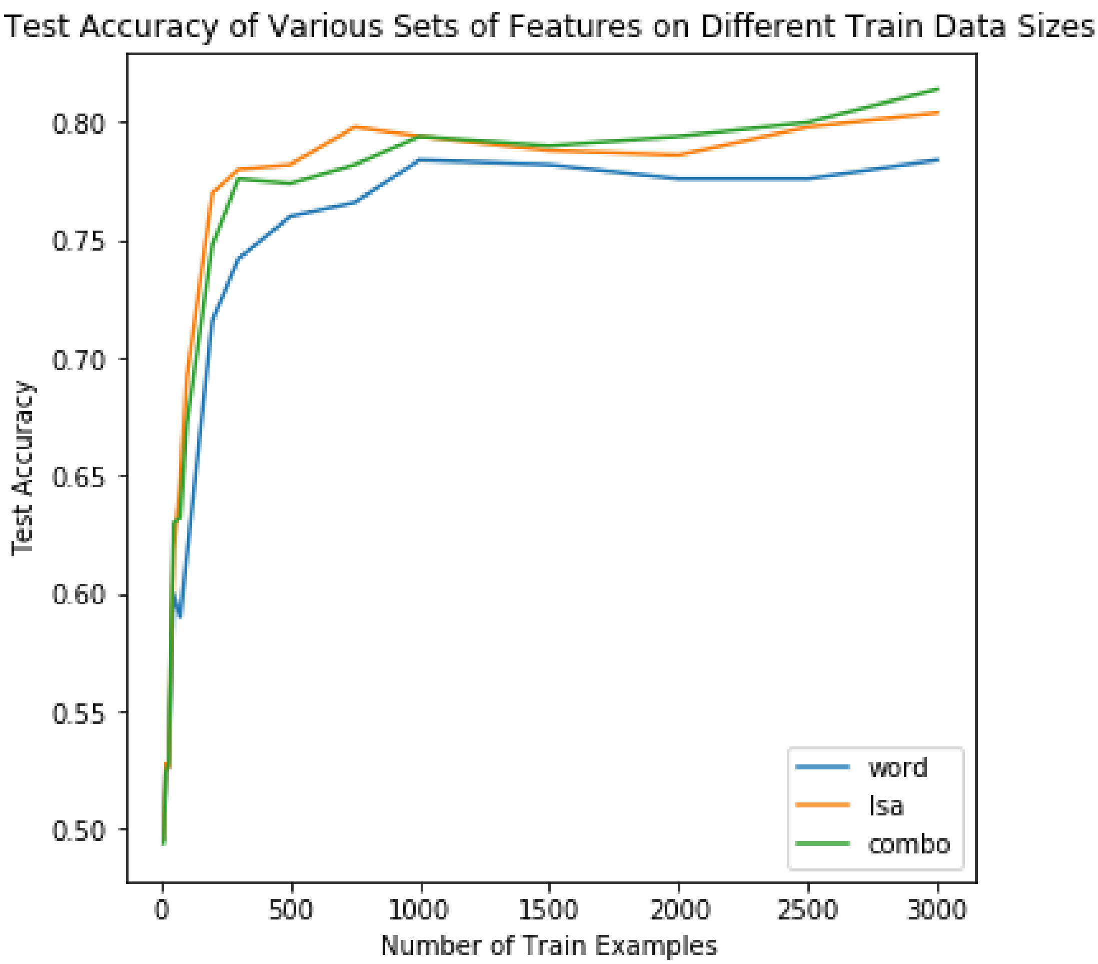
```
lsa features, 75 examples
test accuracy 0.646

combo features, 75 examples
test accuracy 0.632

word features, 100 examples
test accuracy 0.616

lsa features, 100 examples
test accuracy 0.692

combo features, 100 examples
test accuracy 0.672

word features, 200 examples
test accuracy 0.716

lsa features, 200 examples
test accuracy 0.77

combo features, 200 examples
test accuracy 0.748

word features, 300 examples
test accuracy 0.742

lsa features, 300 examples
test accuracy 0.78

combo features, 300 examples
test accuracy 0.776

word features, 500 examples
test accuracy 0.76

lsa features, 500 examples
test accuracy 0.782

combo features, 500 examples
test accuracy 0.774

word features, 750 examples
test accuracy 0.766

lsa features, 750 examples
test accuracy 0.798
```

```
combo features, 750 examples
test accuracy 0.782

word features, 1000 examples
test accuracy 0.784

lsa features, 1000 examples
test accuracy 0.794

combo features, 1000 examples
test accuracy 0.794

word features, 1500 examples
test accuracy 0.782

lsa features, 1500 examples
test accuracy 0.788

combo features, 1500 examples
test accuracy 0.79

word features, 2000 examples
test accuracy 0.776

lsa features, 2000 examples
test accuracy 0.786

combo features, 2000 examples
test accuracy 0.794

word features, 2500 examples
test accuracy 0.776

lsa features, 2500 examples
test accuracy 0.798

combo features, 2500 examples
test accuracy 0.8

word features, 3000 examples
test accuracy 0.784

lsa features, 3000 examples
test accuracy 0.804

combo features, 3000 examples
test accuracy 0.814
```

In [12]:
```
print(max(word_500_results))
print(max(lsa_500_results))
print(max(combo_500_results))
```

```
0.784
0.804
0.814
```

In [13]:
```
plt.figure(figsize=(6, 6))
plt.plot(train_sizes, word_500_results, label="word")
plt.plot(train_sizes, lsa_500_results, label="lsa")
plt.plot(train_sizes, combo_500_results, label="combo")
plt.legend(loc = "lower right")
plt.title('Test Accuracy of Various Sets of Features on Different Train Data Sizes')
plt.xlabel('Number of Train Examples')
plt.ylabel('Test Accuracy')

# plt.savefig('word_embed_matrix_e500.png')
# files.download('word_embed_matrix_e500.png')
plt.show()
```




```
In [14]: embedding_sizes = [10, 50, 100, 500, 1000, 2006]
         for embed_sz in embedding_sizes:
             print("Embedding size: ", embed_sz)
             reps_tfidf = learn_reps_lsa(td_matrix_tfidf, embed_sz)
             lab_util.show_similar_words(vectorizer.tokenizer, reps_tfidf, show_tokens)
             print()
```

```
Embedding size:  10
good 47
  very 0.207
  . 0.210
  p 0.227
  sure 0.235
  u 0.260
bad 201
  agree 0.240
  entirely 0.274
```

```
  positive 0.274
  forward 0.282
  overly 0.285
cookie 504
  cookies 0.204
  muffins 0.261
  cake 0.262
  tough 0.271
  excellent 0.274
jelly 351
  gifts 0.091
  soups 0.098
  vanilla 0.112
  mixing 0.121
  stuck 0.134
dog 925
  dogs 0.119
  him 0.137
  baby 0.147
  he 0.191
  lamb 0.194
the 36
  have 0.086
  in 0.103
  . 0.122
  be 0.140
  that 0.145
4 292
  1 0.039
  6 0.069
  5 0.112
  protein 0.114
  7 0.124

Embedding size:  50
good 47
  quick 0.594
  everyone 0.673
  decide 0.674
  than 0.713
  better 0.726
bad 201
  expect 0.787
  feeling 0.853
  strange 0.875
  just 0.936
  about 0.943
cookie 504
```

```
  cookies 0.282
  nana's 0.325
  oreos 0.634
  bars 0.708
  shortbread 0.818
jelly 351
  gifts 0.432
  creamer 0.603
  online 0.798
  milk 0.838
  maybe 0.864
dog 925
  foods 0.574
  pet 0.636
  nutritious 0.652
  pets 0.687
  switched 0.719
the 36
  of 0.723
  . 0.775
  in 0.824
  on 0.926
  to 0.930
4 292
  6 0.361
  70 0.542
  1 0.625
  concentrated 0.666
  stevia 0.732

Embedding size:  100
good 47
  everyone 1.078
  lunches 1.089
  as 1.145
  pretty 1.182
  but 1.199
bad 201
  taste 1.038
  strange 1.084
  like 1.152
  myself 1.169
  nasty 1.177
cookie 504
  cookies 0.346
  nana's 0.517
  oreos 0.698
  bars 0.796
```

```
  craving 1.026
jelly 351
  creamer 0.891
  gifts 1.008
  twist 1.044
  packages 1.150
  advertised 1.179
dog 925
  foods 0.996
  switched 1.044
  pet 1.096
  loves 1.147
  appeal 1.150
the 36
  of 0.906
  <unk> 0.976
  . 1.053
  and 1.142
  to 1.194
4 292
  1 0.871
  6 0.879
  70 0.934
  concentrated 0.989
  measure 1.024

Embedding size:  500
good 47
  crazy 1.695
  gerber 1.753
  beat 1.758
  homemade 1.785
  tasting 1.799
bad 201
  disgusting 1.623
  awful 1.713
  positive 1.715
  bland 1.731
  gone 1.736
cookie 504
  nana's 1.103
  moist 1.388
  odd 1.452
  impossible 1.486
  needs 1.509
jelly 351
  twist 1.156
  cardboard 1.211
```

```
  advertised 1.402
  plum 1.447
  sold 1.470
dog 925
  happier 1.641
  earlier 1.658
  foods 1.690
  stays 1.697
  eats 1.704
the 36
  <unk> 1.478
  and 1.578
  . 1.581
  of 1.627
  is 1.632
4 292
  mistake 1.687
  2nd 1.707
  toast 1.708
  table 1.714
  70 1.723

Embedding size:  1000
good 47
  luck 1.874
  a 1.879
  suspect 1.891
  shape 1.899
  reminded 1.906
bad 201
  disgusting 1.772
  touch 1.843
  wild 1.847
  entirely 1.869
  timely 1.872
cookie 504
  nana's 1.543
  moist 1.646
  needs 1.732
  chewy 1.778
  odd 1.782
jelly 351
  twist 1.495
  softer 1.595
  shocked 1.696
  gummy 1.734
  supermarket 1.745
dog 925
```

```
  happier 1.820
  earlier 1.835
  owner 1.853
  eats 1.855
  nutrients 1.869
the 36
  <unk> 1.556
  is 1.653
  suspect 1.715
  . 1.738
  fence 1.768
4 292
  economical 1.837
  mistake 1.839
  total 1.861
  70 1.861
  certainly 1.878

Embedding size:  2006
good 47
  from 2.000
  yes 2.000
  packaging 2.000
  become 2.000
  were 2.000
bad 201
  finish 2.000
  due 2.000
  <unk> 2.000
  below 2.000
  raw 2.000
cookie 504
  agave 2.000
  was 2.000
  somewhat 2.000
  authentic 2.000
  watching 2.000
jelly 351
  greta 2.000
  mail 2.000
  varieties 2.000
  issue 2.000
  consistent 2.000
dog 925
  we 2.000
  baby 2.000
  i'd 2.000
  product 2.000
```

```
  door 2.000
the 36
  i 2.000
  good 2.000
  loaded 2.000
  thinks 2.000
  itself 2.000
4 292
  nicely 2.000
  sauce 2.000
  tired 2.000
  gone 2.000
  future 2.000
```

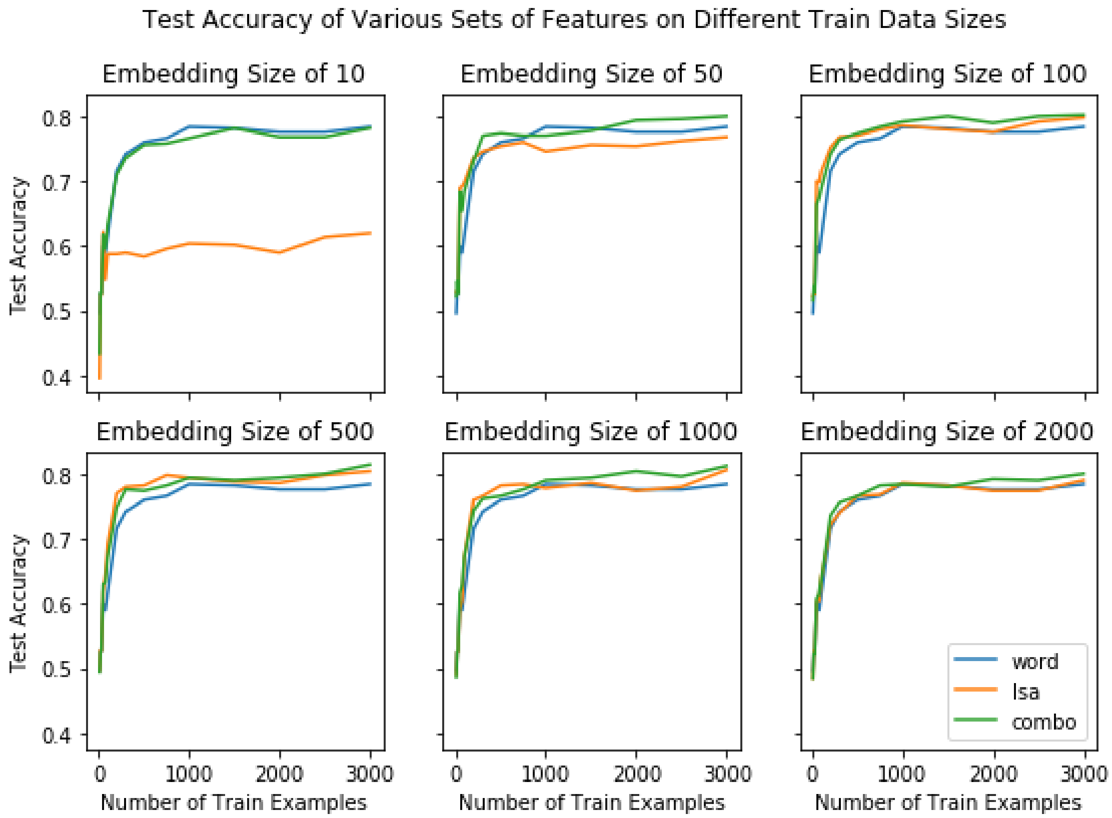
In [0]:
```python
embedding_sizes = [10, 50, 100, 500, 1000, 2000]
train_sizes = [10, 20, 30, 50, 75, 100, 200, 300, 500, 750, 1000, 1500, 2000, 2500, 300

word_embed_results = {esz: {"word": [], "lsa": [], "combo": []} for esz in embedding_si

verbose = False
for embed_sz in embedding_sizes:
    if verbose: print("EMBED SIZE:", embed_sz)
    reps_tfidf = learn_reps_lsa(td_matrix_tfidf, embed_sz)

    word_results = word_embed_results[embed_sz]["word"]
    lsa_results = word_embed_results[embed_sz]["lsa"]
    combo_results = word_embed_results[embed_sz]["combo"]

    for n in train_sizes:
        word_results.append(training_experiment("word", word_featurizer, n, verbose=ve
        lsa_results.append(training_experiment("lsa", lsa_featurizer, n, verbose=verbo
        combo_results.append(training_experiment("combo", combo_featurizer, n, verbose
```

In [34]:
```python
for embed_sz, results in word_embed_results.items():
    print("Embed size:", embed_sz)
    print("word:", word_embed_results[embed_sz]['word'], '>', max(word_embed_results[em
    print("lsa:", word_embed_results[embed_sz]['lsa'], '>', max(word_embed_results[embe
    print("combo:", word_embed_results[embed_sz]['combo'], '>', max(word_embed_results[
```

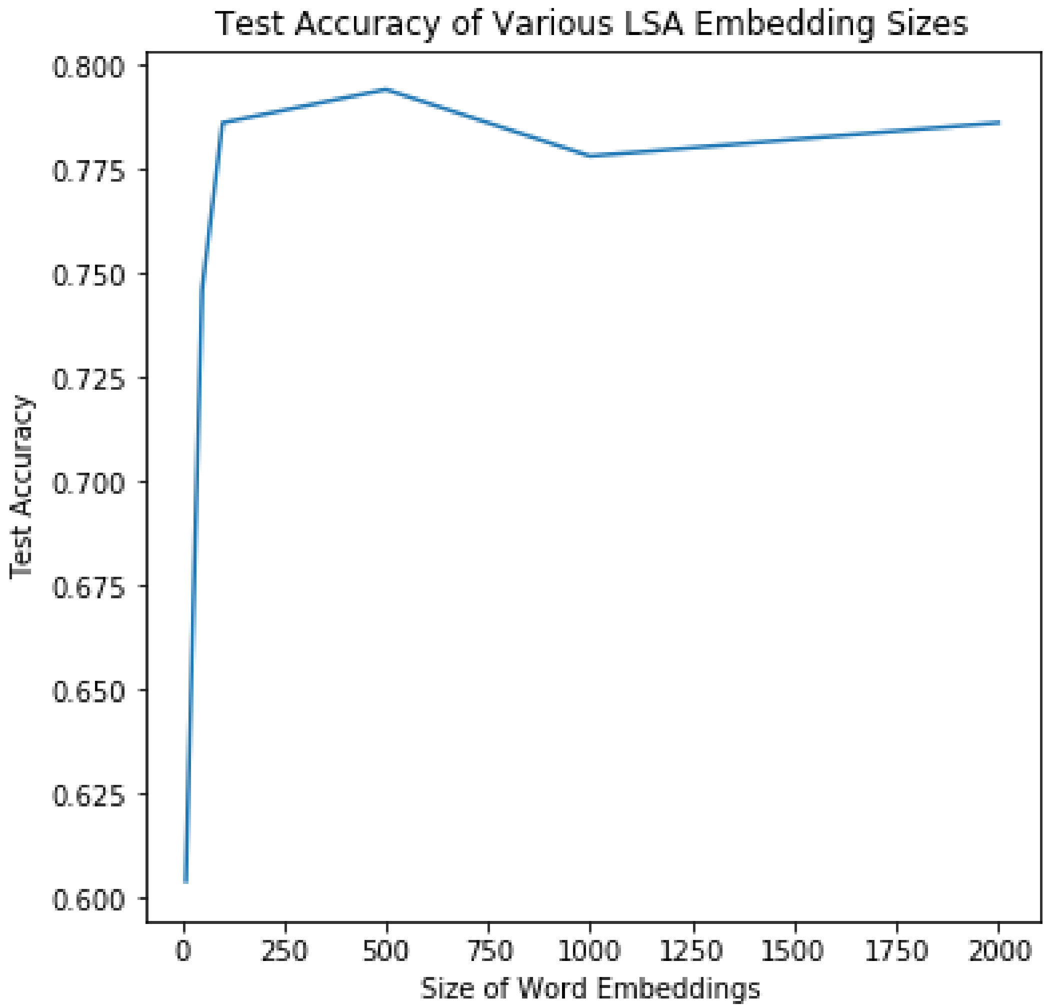
```
Embed size: 10
word: [0.496, 0.526, 0.526, 0.6, 0.59, 0.616, 0.716, 0.742, 0.76, 0.766, 0.784, 0.782, 0.776, 
lsa: [0.396, 0.524, 0.524, 0.622, 0.548, 0.588, 0.588, 0.59, 0.584, 0.596, 0.604, 0.602, 0.59, 
combo: [0.434, 0.528, 0.526, 0.618, 0.596, 0.632, 0.71, 0.736, 0.756, 0.758, 0.766, 0.782, 0.7
Embed size: 50
word: [0.496, 0.526, 0.526, 0.6, 0.59, 0.616, 0.716, 0.742, 0.76, 0.766, 0.784, 0.782, 0.776, 
lsa: [0.528, 0.532, 0.528, 0.69, 0.692, 0.698, 0.736, 0.746, 0.754, 0.76, 0.746, 0.756, 0.754, 
```

```
combo: [0.522, 0.546, 0.526, 0.684, 0.654, 0.684, 0.732, 0.77, 0.774, 0.77, 0.77, 0.778, 0.794
Embed size: 100
word: [0.496, 0.526, 0.526, 0.6, 0.59, 0.616, 0.716, 0.742, 0.76, 0.766, 0.784, 0.782, 0.776, 0
lsa: [0.522, 0.528, 0.526, 0.7, 0.698, 0.716, 0.752, 0.768, 0.77, 0.78, 0.786, 0.78, 0.776, 0.7
combo: [0.516, 0.538, 0.528, 0.664, 0.674, 0.69, 0.742, 0.764, 0.774, 0.784, 0.792, 0.8, 0.79,
Embed size: 500
word: [0.496, 0.526, 0.526, 0.6, 0.59, 0.616, 0.716, 0.742, 0.76, 0.766, 0.784, 0.782, 0.776, 0
lsa: [0.496, 0.528, 0.526, 0.618, 0.646, 0.692, 0.77, 0.78, 0.782, 0.798, 0.794, 0.788, 0.786,
combo: [0.494, 0.524, 0.528, 0.63, 0.632, 0.672, 0.748, 0.776, 0.774, 0.782, 0.794, 0.79, 0.794
Embed size: 1000
word: [0.496, 0.526, 0.526, 0.6, 0.59, 0.616, 0.716, 0.742, 0.76, 0.766, 0.784, 0.782, 0.776, 0
lsa: [0.49, 0.526, 0.526, 0.592, 0.604, 0.672, 0.76, 0.766, 0.782, 0.784, 0.778, 0.786, 0.774,
combo: [0.486, 0.524, 0.528, 0.616, 0.628, 0.672, 0.744, 0.762, 0.766, 0.776, 0.79, 0.794, 0.80
Embed size: 2000
word: [0.496, 0.526, 0.526, 0.6, 0.59, 0.616, 0.716, 0.742, 0.76, 0.766, 0.784, 0.782, 0.776, 0
lsa: [0.484, 0.526, 0.526, 0.604, 0.604, 0.64, 0.722, 0.74, 0.766, 0.768, 0.786, 0.782, 0.774,
combo: [0.484, 0.522, 0.522, 0.608, 0.614, 0.638, 0.736, 0.756, 0.766, 0.782, 0.784, 0.78, 0.79
```

In [22]:
```python
nrow = 2
ncol = 3
fig, axs = plt.subplots(nrow, ncol, figsize=(9,6), sharex = True, sharey=True)
for i, embed_sz in enumerate(embedding_sizes):
    axs[i//ncol, i%ncol].plot(train_sizes, word_embed_results[embed_sz]["word"], label
    axs[i//ncol, i%ncol].plot(train_sizes, word_embed_results[embed_sz]["lsa"], label=
    axs[i//ncol, i%ncol].plot(train_sizes, word_embed_results[embed_sz]["combo"], labe
    axs[i//ncol, i%ncol].set_title('Embedding Size of {}'.format(embed_sz))

for ax in axs.flat:
    ax.set(xlabel='Number of Train Examples', ylabel='Test Accuracy')

# Hide x labels and tick labels for top plots and y ticks for right plots.
for ax in axs.flat:
    ax.label_outer()

plt.legend(loc = "lower right")
plt.suptitle('Test Accuracy of Various Sets of Features on Different Train Data Sizes
# plt.subplots_adjust(hspace=0.2)

# plt.savefig('assets/word_embed_matrix_embed-ntrain.png')
plt.show()
```

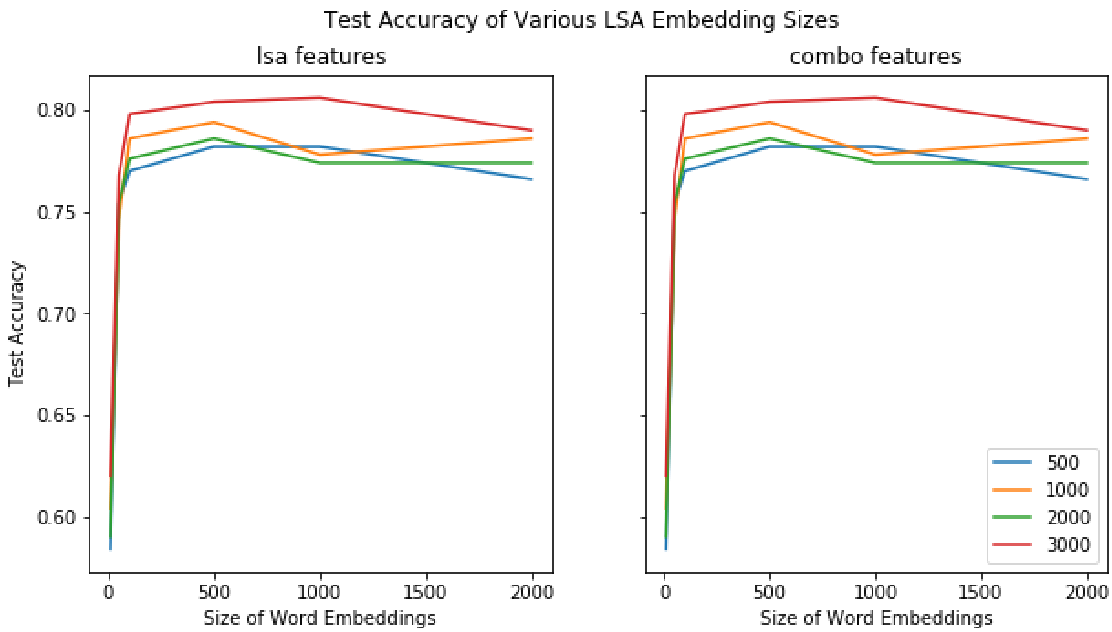



```
In [23]: train_size = 1000
         train_idx = train_sizes.index(train_size)

         lsa_acc_by_embed_size = []
         for embed_sz in embedding_sizes:
             acc = word_embed_results[embed_sz]["lsa"][train_idx]
             lsa_acc_by_embed_size.append(acc)

         plt.figure(figsize=(6, 6))

         plt.plot(embedding_sizes, lsa_acc_by_embed_size, label="lsa")
         # plt.legend(loc = "lower right")
         plt.title('Test Accuracy of Various LSA Embedding Sizes')
         plt.xlabel('Size of Word Embeddings')
         plt.ylabel('Test Accuracy')

         # plt.savefig('assets/word_embed_matrix_ntrain{}.png'.format(train_size))
         plt.show()
```



In [27]:
```
fig, axs = plt.subplots(1, 2, figsize=(10, 5), sharey=True)
ax_lsa, ax_combo = axs

ax_lsa.set_title('lsa features')
ax_combo.set_title('combo features')

for train_size in [500, 1000, 2000, 3000]:
    train_idx = train_sizes.index(train_size)
    lsa_acc_by_embed_size = []
    combo_acc_by_embed_size = []
    for embed_sz in embedding_sizes:
        acc_lsa = word_embed_results[embed_sz]["lsa"][train_idx]
        lsa_acc_by_embed_size.append(acc_lsa)
        acc_combo = word_embed_results[embed_sz]["combo"][train_idx]
        combo_acc_by_embed_size.append(acc_lsa)
    ax_lsa.plot(embedding_sizes, lsa_acc_by_embed_size, label="{}".format(train_size)
```

```
        ax_combo.plot(embedding_sizes, combo_acc_by_embed_size, label="{}".format(train_s

    plt.legend(loc = "lower right")
    plt.suptitle('Test Accuracy of Various LSA Embedding Sizes')

    for ax in axs.flat:
        ax.set(xlabel='Size of Word Embeddings', ylabel='Test Accuracy')
        # Hide x labels and tick labels for top plots and y ticks for right plots.
        ax.label_outer()

    # plt.savefig('assets/word_embed_matrix_ntrain.png')
    plt.show()
```




```
In [25]: u_td, sigma_td, _ = svd(td_matrix)
         u_tt, _, _ = svd(td_matrix@td_matrix.T)
         print(u_td)
         print(u_tt)
```

```
[[ 6.89031937e-01  6.72372892e-01  1.78411037e-01 ...  5.13700980e-05
   1.36794723e-21 -3.92241800e-20]
 [ 2.31553598e-02 -5.20963554e-02 -1.21153239e-02 ... -1.71082540e-05
  -6.62031363e-18 -1.47116322e-18]
 [ 3.83301968e-02 -4.29225155e-02 -1.93856196e-02 ...  4.32473026e-04
   9.83299612e-18  4.67188690e-18]
 ...
 [ 3.17231578e-04 -6.50245704e-04 -1.52675912e-03 ...  2.39091962e-03
  -5.42117020e-17 -5.93870946e-17]
```

```
 [ 4.42263586e-04 -2.14287700e-04  4.98282343e-04 ... -1.15934843e-02
   2.13613803e-16  1.16716819e-16]
 [ 2.76302849e-03  1.39797955e-02  5.86256685e-03 ...  9.53740138e-03
   1.67355993e-17  3.48104400e-16]]
[[-6.89031937e-01  6.72372892e-01  1.78411037e-01 ...  5.13700980e-05
   3.35427360e-18  3.03527002e-19]
 [-2.31553598e-02 -5.20963554e-02 -1.21153239e-02 ... -1.71082541e-05
  -2.52381903e-17  2.77028285e-17]
 [-3.83301968e-02 -4.29225155e-02 -1.93856196e-02 ...  4.32473026e-04
   2.22627435e-17 -1.87185506e-17]
 ...
 [-3.17231578e-04 -6.50245704e-04 -1.52675912e-03 ...  2.39091962e-03
   2.67934253e-16  1.00579829e-16]
 [-4.42263586e-04 -2.14287700e-04  4.98282343e-04 ... -1.15934843e-02
  -1.59470653e-16 -4.69572539e-16]
 [-2.76302849e-03  1.39797955e-02  5.86256685e-03 ...  9.53740138e-03
   1.66122301e-15  1.97027228e-16]]
```

```
In [26]: print(np.diff(sigma_td) <= 0) # confirm that sigma is indecreasing (non-incr.) order
```

```
[ True  True  True ...  True  True  True]
```

0.3 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 matrix factorization objective. This requires a little more work; we've provided scaffolding for a PyTorch model implementation below. (If you've never used PyTorch before, there are some tutorials here. You're also welcome to implement these experiments in any other framework of your choosing.)

```
In [0]: 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__()

              # Your code here!
              self.vocab_size = vocab_size
              self.embed_dim = embed_dim
```

```
        self.V = nn.Embedding(vocab_size, embed_dim)
        self.U = nn.Linear(embed_dim, vocab_size)
        self.log_softmax = nn.LogSoftmax(dim=1)

    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!
        n_batch, n_context = context.shape
        embeddings = self.V(context)  # n_batch x n_context x self.embed_dim
        avg_context = torch.sum(embeddings, dim=1)
        output = self.U(avg_context)
        if not self.training:
          # training uses loss that incorporates softmax
          # apply softmax for prediction
          output = self.log_softmax(output)
        return output
```

```
In [0]: 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()

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

          model.train()
          for epoch in range(n_epochs):
            for context, label in loader:
              # as described above, `context` is a batch of context word ids (n_batch, n_conte
              # `label` is a batch of predicted word labels of shape (n_batch,)
              context = context.to(device)
              label = label.to(device)

              model.zero_grad()  # clear gradients
```

```
            preds = model(context)  # n_batch x vocab_size
            loss = loss_fn(preds, label)
            loss.backward()  # compute gradients
            opt.step()

        # reminder: you want to return a `vocab_size x embedding_size` numpy array
        embedding_matrix = model.V.weight.cpu().detach().numpy()
        return embedding_matrix
```

```
In [0]: reps_word2vec = learn_reps_word2vec(train_reviews, 2, 500, 10, 100)
```

After training the embeddings, we can try to visualize the embedding space to see if it makes sense. First, we can take any word in the space and check its closest neighbors.

```
In [38]: lab_util.show_similar_words(vectorizer.tokenizer, reps_word2vec, show_tokens)
```

```
good 47
  great 1.693
  terms 1.718
  terrible 1.730
  bad 1.736
  prepared 1.736
bad 201
  funny 1.715
  limited 1.717
  stores 1.721
  ready 1.731
  good 1.736
cookie 504
  equal 1.649
  higher 1.704
  dark 1.730
  around 1.732
  gone 1.733
jelly 351
  muffin 1.714
  bears 1.715
  first 1.723
  mixes 1.733
  stuck 1.742
dog 925
  baby 1.648
  rica 1.672
  vanilla 1.693
  introduced 1.717
  bother 1.731
the 36
  a 1.585
  my 1.619
```

```
  their 1.670
  your 1.682
  impossible 1.721
4 292
  unit 1.687
  150 1.700
  sticky 1.706
  three 1.723
  since 1.733
```

We can also cluster the embedding space. Clustering in 4 or more dimensions is hard to visualize, and even clustering in 2 or 3 can be difficult because there are so many words in the vocabulary. One thing we can try to do is assign cluster labels and qualitiatively look for an underlying pattern in the clusters.

```
In [39]: 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}")
```

```
below: 0
general: 0
moved: 1
spread: 1
eaten: 1
without: 1
reviews: 1
suggest: 1
worked: 1
help: 1
given: 1
bought: 1
him: 1
has: 1
needed: 1
will: 1
zero: 1
rate: 1
doesn't: 1
decide: 1
kind: 4
```

```
seen: 6
buying: 7
caramels: 7
than: 7
large: 7
muffin: 7
brewer: 7
hint: 7
lunches: 7
classic: 7
crackers: 7
expiration: 7
nearly: 7
potassium: 7
its: 7
coffee: 7
unfortunately: 7
never: 7
shipment: 7
beef: 7
pouch: 7
fruit: 7
granted: 7
lasts: 7
that's: 7
im: 8
fall: 8
teeth: 8
pieces: 8
amazing: 8
prime: 8
cubes: 8
beer: 8
months: 8
excellent: 8
subtle: 8
filled: 8
lays: 8
someone: 8
ok: 8
caffeine: 8
average: 8
bad: 8
go: 8
disappointed: 8
update: 8
warning: 8
artificial: 8
```

```
times: 8
living: 8
birthday: 8
plus: 8
40: 8
plum: 8
target: 8
un: 8
learned: 8
solid: 8
calcium: 8
treats: 8
mean: 8
puppy: 8
something: 8
fiber: 8
morning: 8
description: 8
still: 8
truly: 8
we: 8
mill: 8
colors: 8
packing: 8
starbucks: 8
double: 8
holes: 8
rica: 8
next: 8
change: 9
cookies: 9
```

Finally, we can use the trained word embeddings to construct vector representations of full reviews. One common approach is to simply average all the word embeddings in the review to create an overall embedding. Implement the transform function in Word2VecFeaturizer to do this.

```
In [40]: def word2vec_lsa_featurizer(xs):
             feats = xs@reps_word2vec # Your code here!

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

         training_experiment("word2vec", word2vec_lsa_featurizer, 10)
```

```
word2vec features, 10 examples
test accuracy 0.526
```

Out[40]: 0.526

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?

In [41]:
```
context_sizes = [1, 2, 3, 5, 10, 20, 30, 50]
for csz in context_sizes:
  print("Context window size: ", csz)
  reps_word2vec_test = learn_reps_word2vec(train_reviews, csz, 500, 10, 100)
  lab_util.show_similar_words(vectorizer.tokenizer, reps_word2vec_test, show_tokens)
  print()
```

```
Context window size:  1
good 47
  switch 1.684
  tolerate 1.693
  salty 1.695
  significant 1.711
  chemical 1.711
bad 201
  greta 1.684
  harder 1.695
  they 1.709
  plant 1.709
  options 1.713
cookie 504
  pouch 1.702
  shot 1.704
  across 1.709
  mustard 1.720
  waste 1.721
jelly 351
  exact 1.683
```

```
  peanuts 1.689
  veggies 1.689
  same 1.698
  design 1.705
dog 925
  bed 1.613
  carton 1.708
  life 1.713
  junk 1.722
  times 1.728
the 36
  my 1.485
  your 1.595
  a 1.631
  wise 1.685
  those 1.719
4 292
  small 1.667
  inches 1.722
  bed 1.729
  caffeine 1.730
  yellow 1.731

Context window size:  2
good 47
  example 1.661
  sojos 1.677
  hair 1.690
  picked 1.705
  dark 1.712
bad 201
  watchers 1.722
  www 1.725
  burn 1.727
  strange 1.728
  rest 1.747
cookie 504
  pork 1.624
  tiny 1.675
  overpriced 1.684
  clams 1.697
  seed 1.700
jelly 351
  shown 1.567
  egg 1.692
  older 1.693
  zero 1.711
  dishes 1.718
```

```
dog 925
  breed 1.638
  roast 1.652
  daily 1.707
  ground 1.733
  packages 1.740
the 36
  a 1.554
  my 1.635
  their 1.682
  your 1.683
  brewers 1.729
4 292
  flavour 1.679
  0 1.687
  okay 1.706
  stock 1.729
  added 1.733

Context window size:  3
good 47
  excited 1.678
  rich 1.693
  based 1.726
  choose 1.731
  boost 1.737
bad 201
  yeah 1.700
  first 1.733
  strong 1.736
  wonderful 1.743
  mint 1.745
cookie 504
  beef 1.682
  picture 1.703
  machine 1.732
  spinach 1.737
  aren't 1.737
jelly 351
  somewhere 1.692
  california 1.714
  expecting 1.727
  grains 1.747
  rancid 1.749
dog 925
  worse 1.692
  candy 1.714
  popcorn 1.715
```

```
  warehouse 1.720
  felidae 1.737
the 36
  a 1.505
  my 1.633
  amazon's 1.645
  this 1.680
  grown 1.747
4 292
  24 1.676
  vomiting 1.728
  pork 1.729
  25 1.731
  amazing 1.739

Context window size:  5
good 47
  break 1.735
  great 1.737
  serious 1.743
  watery 1.757
  due 1.759
bad 201
  tight 1.686
  tasty 1.713
  healthier 1.746
  pass 1.748
  berry 1.756
cookie 504
  hopes 1.677
  beverages 1.690
  meant 1.721
  usually 1.726
  frosting 1.733
jelly 351
  mueslix 1.660
  mold 1.718
  bunch 1.722
  generally 1.734
  thrown 1.745
dog 925
  garbage 1.605
  burn 1.641
  kids 1.678
  coffees 1.693
  recent 1.706
the 36
  a 1.409
```

```
  this 1.695
  particular 1.723
  tart 1.739
  it 1.746
4 292
  ground 1.739
  nutritious 1.748
  bisquick 1.748
  the 1.756
  it 1.758

Context window size:  10
good 47
  noodles 1.669
  german 1.707
  readily 1.736
  candy 1.746
  peach 1.753
bad 201
  kona 1.700
  twice 1.715
  hard 1.717
  worry 1.734
  caramel 1.748
cookie 504
  guy 1.671
  melted 1.677
  pan 1.694
  glass 1.704
  attention 1.707
jelly 351
  soy 1.692
  plan 1.701
  note 1.710
  often 1.741
  vegetables 1.743
dog 925
  combination 1.714
  effects 1.717
  oil 1.718
  alone 1.758
  stuck 1.758
the 36
  a 1.379
  my 1.489
  this 1.614
  it 1.634
  these 1.702
```

```
4 292
  milder 1.712
  moved 1.732
  batch 1.741
  numerous 1.746
  treat 1.747

Context window size:  20
good 47
  rip 1.678
  fine 1.702
  same 1.729
  kinda 1.734
  dont 1.758
bad 201
  seen 1.658
  couple 1.733
  holes 1.740
  follow 1.753
  face 1.755
cookie 504
  further 1.720
  izze 1.735
  room 1.737
  three 1.737
  metallic 1.749
jelly 351
  purchased 1.704
  list 1.734
  safe 1.742
  problem 1.749
  my 1.753
dog 925
  cat 1.729
  items 1.740
  split 1.740
  while 1.749
  along 1.753
the 36
  a 1.258
  this 1.426
  it 1.499
  another 1.663
  blends 1.685
4 292
  recipes 1.693
  meat 1.709
  lemon 1.725
```

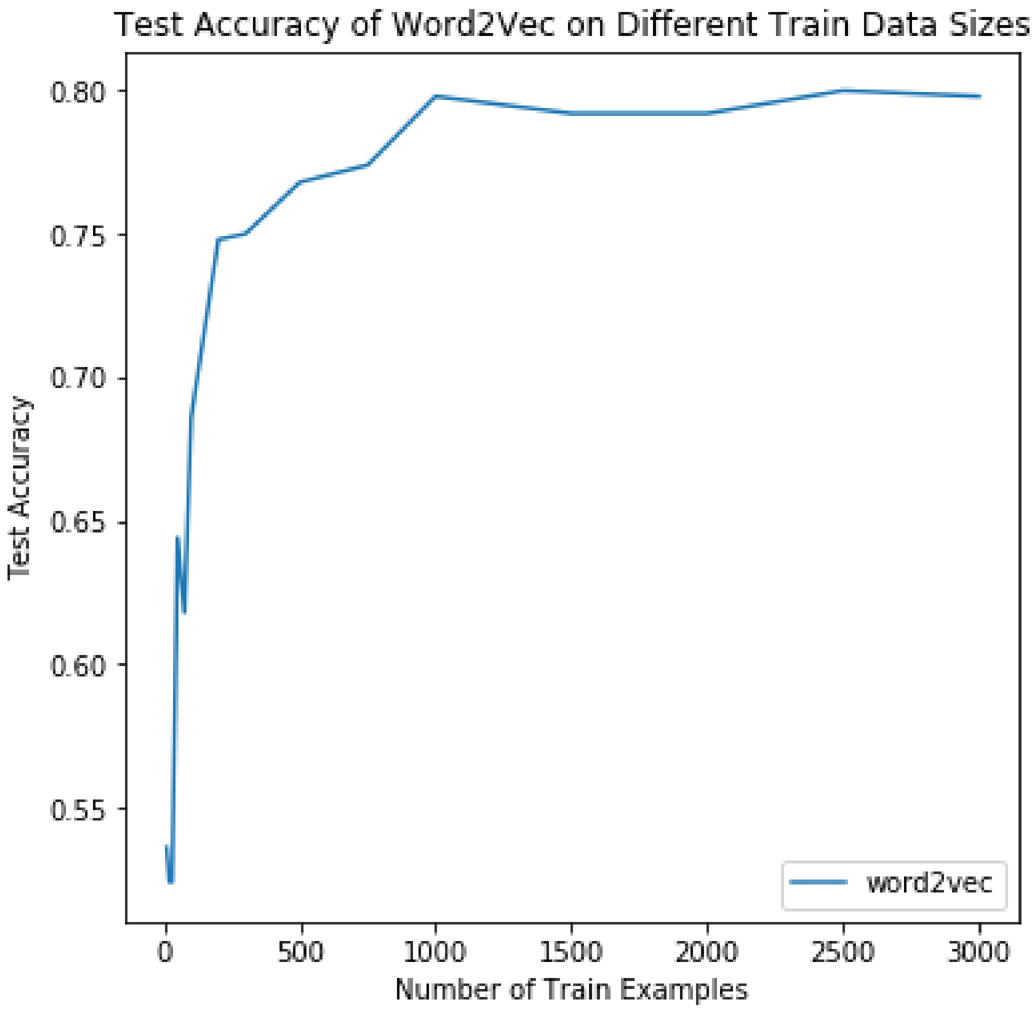
```
  basically 1.756
  someone 1.758

Context window size:  30
good 47
  miss 1.728
  package 1.745
  rich 1.748
  version 1.748
  great 1.753
bad 201
  offer 1.642
  much 1.738
  iron 1.741
  bone 1.747
  further 1.748
cookie 504
  oven 1.714
  later 1.731
  carbonation 1.736
  vomiting 1.749
  pudding 1.763
jelly 351
  tangerine 1.700
  from 1.713
  kind 1.724
  im 1.725
  bills 1.742
dog 925
  cat 1.629
  brewer 1.696
  sweet 1.735
  paying 1.737
  iams 1.742
the 36
  a 1.482
  this 1.582
  my 1.667
  it 1.674
  granted 1.688
4 292
  range 1.707
  plum 1.730
  cheddar 1.731
  contacted 1.732
  grounds 1.746

Context window size:  50
```

```
good 47
  sell 1.718
  great 1.730
  clearly 1.731
  thick 1.738
  stronger 1.745
bad 201
  sell 1.716
  processed 1.717
  plan 1.738
  office 1.739
  bigger 1.741
cookie 504
  packaging 1.707
  carbs 1.710
  doubt 1.723
  caffeine 1.724
  required 1.725
jelly 351
  outside 1.664
  30 1.681
  leaf 1.701
  watermelon 1.714
  brewer 1.718
dog 925
  amount 1.682
  vet 1.690
  problems 1.721
  sweeter 1.722
  pineapple 1.723
the 36
  a 1.435
  your 1.596
  it 1.673
  dairy 1.675
  comparison 1.698
4 292
  risk 1.678
  gum 1.697
  lots 1.716
  stick 1.720
  an 1.731
```

In [0]:
```
train_sizes = [10, 20, 30, 50, 75, 100, 200, 300, 500, 750, 1000, 1500, 2000, 2500, 300

word2vec_500_results = []
```

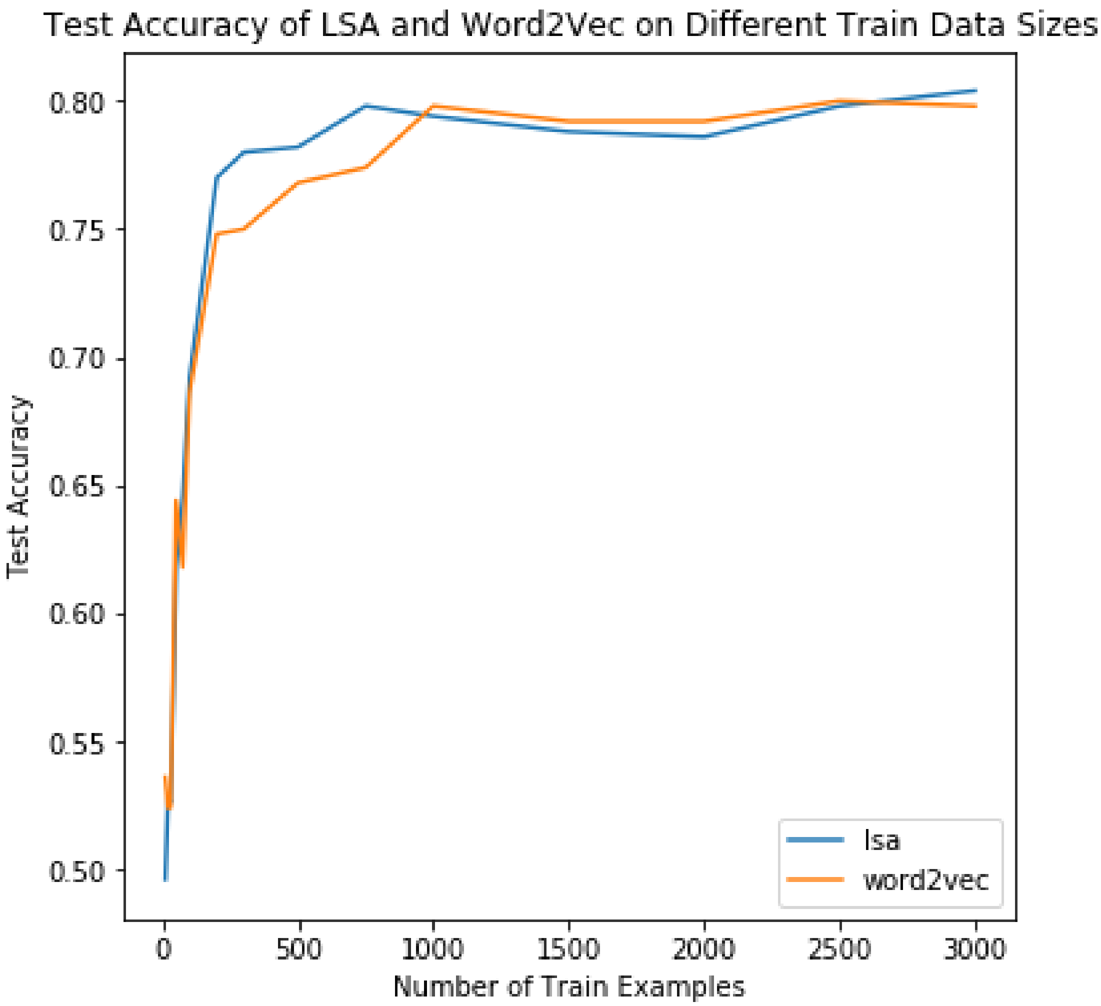
```
reps_word2vec = learn_reps_word2vec(train_reviews, 2, 500, 10, 100)

for n in train_sizes:
    word2vec_500_results.append(training_experiment("word2vec", word2vec_lsa_featurize
```

In [43]:
```
plt.figure(figsize=(6, 6))
plt.plot(train_sizes, word2vec_500_results, label="word2vec")
plt.legend(loc = "lower right")
plt.title('Test Accuracy of Word2Vec on Different Train Data Sizes')
plt.xlabel('Number of Train Examples')
plt.ylabel('Test Accuracy')

# plt.savefig('word2vec_e500.png')
plt.show()
```



```
In [0]: # files.download('word2vec_e500.png')
```

```
In [46]: plt.figure(figsize=(6, 6))
         plt.plot(train_sizes, lsa_500_results, label="lsa")
         plt.plot(train_sizes, word2vec_500_results, label="word2vec")
         plt.legend(loc = "lower right")
         plt.title('Test Accuracy of LSA and Word2Vec on Different Train Data Sizes')
         plt.xlabel('Number of Train Examples')
         plt.ylabel('Test Accuracy')

         plt.savefig('word2vec-vs-lsa_e500.png')
         plt.show()
```

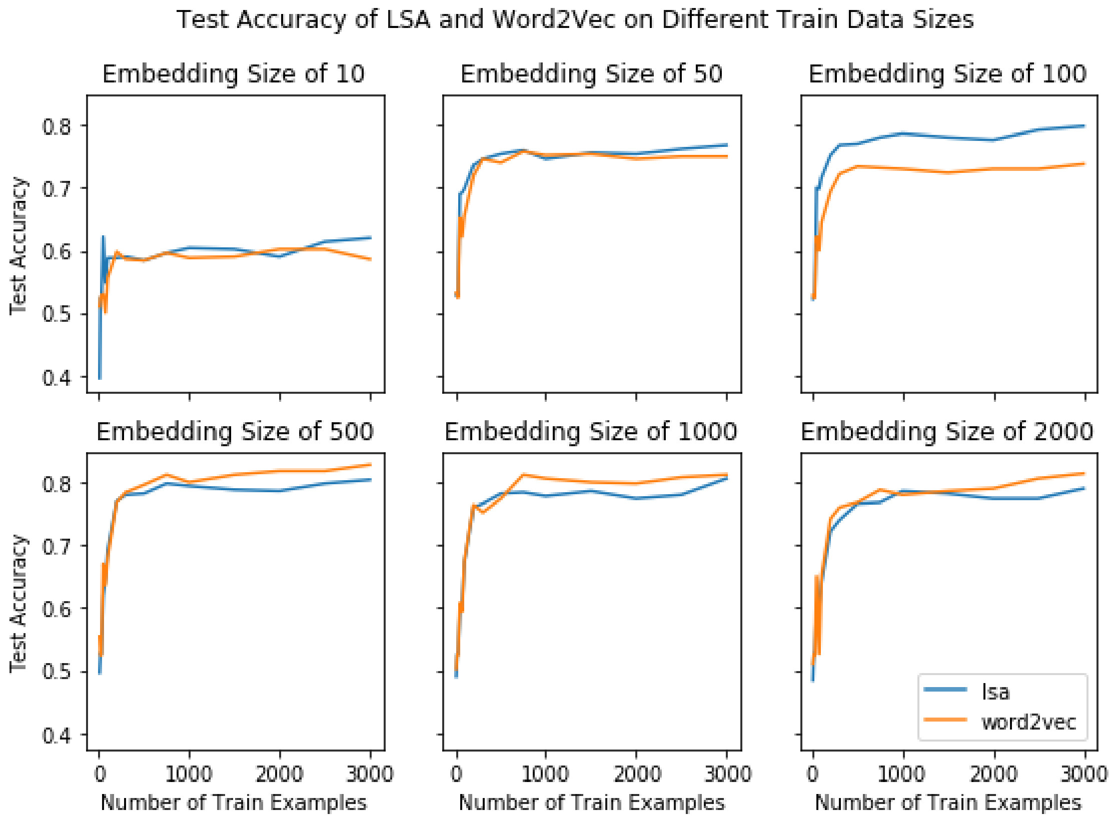



```
In [0]: # files.download('word2vec-vs-lsa_e500.png')
```

```
In [0]: embedding_sizes = [10, 50, 100, 500, 1000, 2000]
        train_sizes = [10, 20, 30, 50, 75, 100, 200, 300, 500, 750, 1000, 1500, 2000, 2500, 30
```

```
word2vec_results = {esz: [] for esz in embedding_sizes}

for embed_sz in embedding_sizes:
    reps_word2vec = learn_reps_word2vec(train_reviews, 2, embed_sz, 10, 100)

    for n in train_sizes:
        word2vec_results[embed_sz].append(training_experiment("word2vec", word2vec_lsa
```

In [50]:
```
nrow = 2
ncol = 3
fig, axs = plt.subplots(nrow, ncol, figsize=(9,6), sharex = True, sharey=True)
for i, embed_sz in enumerate(embedding_sizes):
    axs[i//ncol, i%ncol].plot(train_sizes, word_embed_results[embed_sz]["lsa"], label=
    axs[i//ncol, i%ncol].plot(train_sizes, word2vec_results[embed_sz], label="word2vec
    axs[i//ncol, i%ncol].set_title('Embedding Size of {}'.format(embed_sz))

for ax in axs.flat:
    ax.set(xlabel='Number of Train Examples', ylabel='Test Accuracy')

# Hide x labels and tick labels for top plots and y ticks for right plots.
for ax in axs.flat:
    ax.label_outer()

plt.legend(loc = "lower right")
plt.suptitle('Test Accuracy of LSA and Word2Vec on Different Train Data Sizes')
# plt.subplots_adjust(hspace=0.2)

# plt.savefig('lsa-word2vec_embed-ntrain.png')
# files.download('lsa-word2vec_embed-ntrain.png')
plt.show()
```


Test Accuracy of LSA and Word2Vec on Different Train Data Sizes
Embedding Size of 10
Embedding Size of 50
Embedding Size of 100
Embedding Size of 500
Embedding Size of 1000
Embedding Size of 2000
Test Accuracy
0.4
0.5
0.6
0.7
0.8
0
1000
2000
3000
Number of Train Examples
lsa
word2vec

6864_hw1b

March 1, 2020

```
In [1]: %%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'...
```

```
In [0]: 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
```

```
In [0]: from matplotlib import pyplot as plt
```

0.1 Hidden Markov Models

In the remaining part of the lab (containing part 3) you'll use the Baum–Welch algorithm to learn *categorical* representations of words in your 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:

```
In [4]: 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 found them all
rating: 1 (good)

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

review: This is a confection that has been around a few centuries.  It is a light, pillowy citr
rating: 1 (good)

review: If you are looking for the secret ingredient in Robitussin I believe I have found it.
rating: 0 (bad)

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

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

Read 4000 total reviews.
```

0.2 HMM Implementation

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 underflow errors. You'll probably want to work in the log domain (remember that `log(ab) = log(a) + log(b)`, `log(a+b) = logaddexp(a, b)`).

In [0]:
```python
import numpy as np
from scipy.special import logsumexp


# hmm model
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)

        self.eps = 1e-20  # small value for log probs of 0, i.e for beta_T

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

        # 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
        self.B = np.random.rand(num_states, num_words)
        self.B = self.B / np.sum(self.B, axis=1, keepdims=True)

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

        log_alpha = np.zeros((len(obs), self.num_states))

        # your code here!
        # First time step: alpha_0(i) = pi_i B_i(o_0) for 0<=i<N
        # in logspace: log(alpha[0, i]) = log(pi_i) + log(B[i, o_0])
        log_alpha[0, :] = np.log(self.B[:, obs[0]]) + np.log(self.pi)

        # Further time steps: alpha ...
        for t in range(1, len(obs)):
            # log(alpha[t-1,i]) + log(self.A[i, j])
            # sum should match index in log_alpha with row index in self.A
            # so the term from log_alpha should broadcast one elem to each row
            log_trans_prob = np.log(self.A) + log_alpha[t-1, :][:, None]
            log_trans_prob_over_prevs = logsumexp(log_trans_prob, axis=0)  # sum over
            log_alpha[t, :] = log_trans_prob_over_prevs + np.log(self.B[:, obs[t]])

        return log_alpha

    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)

        log_beta = np.zeros((len(obs), self.num_states))
        T = len(obs)

        # beta for last time step is 1, log of which is 0
        # log_beta[T-1, :] = np.zeros((self.num_states,))

        # Further time steps: beta[t-1,i] = sum(j=0 to N-1) A[i,j] B[j,o_t] beta[t,j]
        for t in range(T-1, 0, -1):  # t is the time step for the future obs (=t+1 in
            # add same value to each column (constant j value)
```

```
            log_trans_prob = np.log(self.A) + np.log(self.B[:, obs[t]][None, :]) + log_
            log_beta[t-1, :] = logsumexp(log_trans_prob, axis=1)

        return log_beta

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

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

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

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

        log_alpha = self.forward(obs)  # T=len(obs) x num_states
        log_beta = self.backward(obs)  # T x num_states

        logprob = logsumexp(log_alpha[len(obs)-1, :])

        logprob_backward = logsumexp(np.log(self.pi) + np.log(self.B[:, obs[0]]) + log_

        # Compute log_xi
        # log_xi[t,i,j] = log_alpha[t,i] + np.log(self.A[i,j]) + np.log(self.B[j,obs[t
        log_xi = np.zeros((len(obs), self.num_states, self.num_states))

        # shift B and beta to have info from timestep t+1 in index t
        relevant_B = np.hstack((self.B[:, obs[1:]], np.ones((self.num_states, 1))))
        relevant_logbeta = np.vstack((log_beta[1:, :], np.ones((1, self.num_states))))

        # change dimensions of the four matrices for broadcasting
        new_log_A = np.tile(np.expand_dims(np.log(self.A), axis=0), (len(obs), 1, 1))
        new_log_alpha = np.expand_dims(log_alpha, axis=2)
        new_log_B = np.expand_dims(np.log(relevant_B.T), axis=1)
        new_log_beta = np.expand_dims(relevant_logbeta, axis=1)

        # print(new_log_A.shape)
        # print(new_log_alpha.shape)
        # print(new_log_B.shape)
        # print(new_log_beta.shape)
        log_xi = new_log_A + new_log_alpha + new_log_B + new_log_beta - logprob

        # Compute log_gamma
        log_gamma = np.zeros((len(obs), self.num_states))
        log_gamma = log_alpha + log_beta - logprob
```

```
        return logprob, log_xi, log_gamma

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

        for i_iter in range(num_iters):
            expected_si = np.full((self.num_states, ), self.eps)  # E(si -> s*): shape
            expected_sij = np.full((self.num_states, self.num_states), self.eps)  # E(
            expected_sj = np.full((self.num_states,), self.eps)  # E(sj): shape (num_s
            expected_sjwk = np.full((self.num_states, self.num_words), self.eps)  # E(
            total_logprob = 0
            for i, review in enumerate(corpus):
                logprob, log_xi, log_gamma = self.forward_backward(review)
                # your code here
                total_logprob += logprob

                words_onehot = np.eye(self.num_words)[review]
                max_log_gamma = np.max(log_gamma)
                simplified_gamma = np.exp(log_gamma - max_log_gamma)
                simp_gamma_by_word = simplified_gamma.T @ words_onehot + self.eps  # a
                log_gamma_by_word = np.log(simp_gamma_by_word) + max_log_gamma

                if i == 0:
                    expected_si = logsumexp(log_gamma[0:-1], axis=0)
                    expected_sij = logsumexp(log_xi[0:-1], axis=0)
                    expected_sj = logsumexp(log_gamma, axis=0)
                    expected_sjwk = log_gamma_by_word
                else:
                    np.logaddexp(expected_si, logsumexp(log_gamma[0:-1], axis=0), out=e
                    np.logaddexp(expected_sij, logsumexp(log_xi[0:-1], axis=0), out=exp
                    np.logaddexp(expected_sj, logsumexp(log_gamma, axis=0), out=expecte

                    np.logaddexp(expected_sjwk, log_gamma_by_word, out=expected_sjwk)

            if verbose: print("log-likelihood", total_logprob)
            # print(expected_sij)
            # print(expected_si)
            # print(expected_sjwk.shape, expected_sj.shape)
            # print(expected_sjwk)
            # print(expected_sj)
            A_new = np.exp(expected_sij - expected_si[:, None])
            B_new = np.exp(expected_sjwk - expected_sj[:, None])
            # print("A_new:", A_new)
            # print(np.sum(A_new, axis=1, keepdims=True))
            # print(B_new.shape)

            self.A = A_new / np.sum(A_new, axis=1, keepdims=True)
```

```
                    self.B = B_new / np.sum(B_new, axis=1, keepdims=True)
```

0.3 Testing

```
In [6]: corpus = np.array([[0,3,0,3,0,3,0,3,0,3,0,3], [0,2,0,2,0,2,0,2,0,2,0,2,0], [1,2,1,2,1,2
        hmm = HMM(num_states=2, num_words=4)
        hmm.learn_unsupervised(corpus, 1000, verbose=False)
        print(np.round(hmm.B, 2))
        print()
        hmm.generate(10)
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:70: RuntimeWarning: divide by zero
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:90: RuntimeWarning: divide by zero
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:123: RuntimeWarning: divide by zer


[[0.   0.5  0.27 0.23]
 [0.52 0.   0.24 0.24]]
```

```
Out[6]: [1, 3, 1, 1, 3, 3, 2, 1, 1, 3]
```

```
In [7]: corpus = np.array([[0,3,0,3,0,3,0,3,0,3,0,3], [0,2,0,2,0,2,0,2,0,2,0,2,0], [1,2,1,2,1,2
        hmm = HMM(num_states=2, num_words=4)

        obs = corpus[0]
        log_alpha = hmm.forward(obs)
        alpha = np.exp(log_alpha)
        print("alpha:", alpha)
        print("pi:", hmm.pi)
        print("B:", hmm.B)

        # print(alpha[0, 0], hmm.pi[0] * hmm.B[0, obs[0]])
        for i in range(hmm.num_states):
          assert np.isclose(alpha[0, i], hmm.pi[i] * hmm.B[i, obs[0]])

        for t in range(1, len(obs)):
          for j in range(hmm.num_states):
            prev_sum = 0
            for i in range(hmm.num_states):
              prev_sum += alpha[t-1, i] * hmm.A[i, j]
            res = hmm.B[j, obs[t]] * prev_sum
            # print(alpha[t,j], res)
            assert np.isclose(alpha[t, j], res)
```

```
alpha: [[2.14839665e-01 1.38796425e-01]
 [1.47412296e-02 6.37276832e-02]
 [8.07696426e-03 2.53086511e-02]
```

```
 [1.67647572e-03 5.54125827e-03]
 [7.27979669e-04 2.35249833e-03]
 [1.55090965e-04 5.10617255e-04]
 [6.71215093e-05 2.17012015e-04]
 [1.43056829e-05 4.70967427e-05]
 [6.19100440e-06 2.00164248e-05]
 [1.31950432e-06 4.34402858e-06]
 [5.71035329e-07 1.84624115e-06]
 [1.21706207e-07 4.00677151e-07]]
pi: [0.71762307 0.28237693]
B: [[0.29937675 0.25750521 0.2933972  0.14972084]
 [0.49152891 0.21118213 0.0475508  0.24973817]]
```

In [8]:
```
corpus = np.array([[0,3,0,3,0,3,0,3,0,3,0,3], [0,2,0,2,0,2,0,2,0,2,0,2,0], [1,2,1,2,1,2
hmm = HMM(num_states=2, num_words=4)

obs = corpus[0]
log_beta = hmm.backward(obs)
beta = np.exp(log_beta)
print("beta:", beta)
# print("pi:", hmm.pi)
# print("B:", hmm.B)

for i in range(hmm.num_states):
  assert np.isclose(beta[len(obs)-1, i], 1)

for t in range(len(obs)-2, -1, -1):
  for i in range(hmm.num_states):
    fut_sum = 0
    for j in range(hmm.num_states):
      fut_sum += hmm.A[i,j] * hmm.B[j, obs[t+1]] * beta[t+1, j]
    assert np.isclose(beta[t, i], fut_sum)
```

```
beta: [[2.70381525e-05 2.98880927e-05]
 [9.17308160e-05 9.48983330e-05]
 [1.72712937e-04 1.90917640e-04]
 [5.85953447e-04 6.06186749e-04]
 [1.10324694e-03 1.21953399e-03]
 [3.74292421e-03 3.87216870e-03]
 [7.04727104e-03 7.79007335e-03]
 [2.39090566e-02 2.47343684e-02]
 [4.50184099e-02 4.97596750e-02]
 [1.52793548e-01 1.57971024e-01]
 [2.88340511e-01 3.17376018e-01]
 [1.00000000e+00 1.00000000e+00]]
```

In [9]:
```
from scipy.special import logsumexp
```

```
corpus = np.array([[0,3,0,3,0,3,0,3,0,3,0,3], [0,2,0,2,0,2,0,2,0,2,0,2,0], [1,2,1,2,1,2
hmm = HMM(num_states=2, num_words=4)

obs = corpus[0]
log_alpha = hmm.forward(obs)
alpha = np.exp(log_alpha)
log_beta = hmm.backward(obs)
beta = np.exp(log_beta)

prob_forward = 0
for i in range(hmm.num_states):
  prob_forward += alpha[len(obs)-1, i]

prob_backward = 0
for i in range(hmm.num_states):
  prob_backward += hmm.pi[i] * hmm.B[i, obs[0]] * beta[0, i]

print(prob_forward, prob_backward)
assert np.isclose(prob_forward, prob_backward)
print(np.log(prob_forward), np.log(prob_backward))

logprob_forward = logsumexp(log_alpha[len(obs)-1, :])
logprob_backward = logsumexp(np.log(hmm.pi) + np.log(hmm.B[:, obs[0]]) + log_beta[0, :

print(logprob_forward)
print(logprob_backward)

assert np.isclose(np.log(prob_forward), logprob_forward)
assert np.isclose(np.log(prob_forward), logprob_backward)
assert np.isclose(np.log(prob_backward), logprob_forward)
assert np.isclose(np.log(prob_backward), logprob_backward)
```

```
2.429158639789488e-07 2.42915863978949e-07
-15.230550692330104 -15.230550692330102
-15.230550692330104
-15.230550692330102
```

In [10]:
```
corpus = np.array([[0,3,0,3,0,3,0,3,0,3,0,3], [0,2,0,2,0,2,0,2,0,2,0,2,0], [1,2,1,2,1
hmm = HMM(num_states=2, num_words=4)

obs = corpus[0]

log_alpha = hmm.forward(obs)  # T=len(obs) x num_states
log_beta = hmm.backward(obs)  # T x num_states

logprob = logsumexp(log_alpha[len(obs)-1, :])
```

```python
log_gamma = np.zeros((len(obs), hmm.num_states))
log_xi = np.zeros((len(obs), hmm.num_states, hmm.num_states))

logprob_backward = logsumexp(np.log(hmm.pi) + np.log(hmm.B[:, obs[0]]) + log_beta[0,

# log_xi[t,i,j] = log_alpha[t,i] + np.log(self.A[i,j]) + np.log(self.B[j,obs[t+1]]) +
relevant_B = np.hstack((hmm.B[:, obs[1:]], np.ones((hmm.num_states, 1))))
# print(relevant_B)
# print(np.log(relevant_B))
relevant_logbeta = np.vstack((log_beta[1:, :], np.zeros((1, hmm.num_states))))
# print(log_beta)
# print(relevant_logbeta)

# print(np.tile(np.expand_dims(np.log(hmm.A), axis=0), (len(obs), 1, 1)).shape)
# print(np.tile(np.expand_dims(np.log(hmm.A), axis=0), len(obs)))
new_log_A = np.tile(np.expand_dims(np.log(hmm.A), axis=0), (len(obs), 1, 1))

new_log_alpha = np.expand_dims(log_alpha, axis=2)
# print(new_log_alpha.shape)
# print(new_log_alpha[0])

# print(new_log_A[0])
# print((new_log_A + new_log_alpha)[0])

new_log_B = np.expand_dims(np.log(relevant_B.T), axis=1)
# print(new_log_B)

new_log_beta = np.expand_dims(relevant_logbeta, axis=1)
# print(new_log_beta.shape)

log_xi = new_log_A + new_log_alpha + new_log_B + new_log_beta - logprob

# log_xi[t,:,:] = np.log(hmm.A) + log_alpha[t, :][:, None] + np.log(relevant_B)[:, t]

t=0
log_expected = np.zeros((hmm.num_states, hmm.num_states))
for i in range(hmm.num_states):
  for j in range(hmm.num_states):
    log_expected[i,j] = np.log(hmm.A[i,j]) + log_alpha[t, i] + np.log(hmm.B[j, obs[t+


print(log_xi[0,:,:])
print(log_expected)
# print("-----")
# print(np.log(hmm.A))
# print(log_alpha[t, :][:, None])
# print(np.log(relevant_B)[:, t][None, :])
# print(relevant_logbeta[t, :][None, :])
```

```
[[-1.20548595 -2.03378722]
 [-1.57088655 -1.01679165]]
[[-1.20548595 -2.03378722]
 [-1.57088655 -1.01679165]]
```

```
In [0]: corpus = np.array([[0,3,0,3,0,3,0,3,0,3,0,3], [0,2,0,2,0,2,0,2,0,2,0,2,0], [1,2,1,2,1,2
        hmm = HMM(num_states=2, num_words=4)

        obs = corpus[0]

        log_alpha = hmm.forward(obs)
        alpha = np.exp(log_alpha)
        logprob_forward = logsumexp(log_alpha[len(obs)-1, :])

        log_beta = hmm.backward(obs)
        beta = np.exp(log_beta)

        logprob, log_xi, log_gamma = hmm.forward_backward(obs)
        xi = np.exp(log_xi)
        gamma = np.exp(log_gamma)

        # make sure logprob matches what we'd expect
        assert np.isclose(logprob, logprob_forward)

        # values for t = T (i.e. len(obs)-1) don't matter for updates
        # full_log_expected = np.zeros((len(obs), hmm.num_states, hmm.num_states))
        for t in range(len(obs)-1):
          for i in range(hmm.num_states):
            for j in range(hmm.num_states):
              expected = alpha[t, i] * hmm.A[i,j] * hmm.B[j, obs[t+1]] * beta[t+1, j] / np.exp
              log_expected = log_alpha[t, i] + np.log(hmm.A[i,j]) + np.log(hmm.B[j, obs[t+1]])
              # full_log_expected[t,i,j] = log_expected
              assert np.isclose(log_xi[t, i, j], log_expected), "Expected log_xi value: {} but
              assert np.isclose(xi[t, i, j], expected), "Expected xi value: {} but got {}".form

        # for t in range(len(obs)):
        #   print('t =', t)
        #   print(log_xi[t])
        #   print(full_log_expected[t])
        #   print('---')

        # skip the last time step, since xi won't match here
        for t in range(len(obs)-1):
          for i in range(hmm.num_states):
            expected = 0
            for j in range(hmm.num_states):
              expected += xi[t,i,j]
```

```
            assert np.isclose(gamma[t,i], expected), "At time {} and state {}, expected {} but

        assert np.allclose(np.sum(gamma, axis=1), 1)
```

In [12]:
```
# Not sure what to set the last column in relevant_B and relevant_logbeta
# to get the right values for the last time step
gamma_from_xi_last_time = np.zeros((hmm.num_states,))
for i in range(hmm.num_states):
  expected = 0
  for j in range(hmm.num_states):
    expected += xi[-1,i,j]
  gamma_from_xi_last_time[i] = expected

print(gamma_from_xi_last_time)
print(gamma[-1, :])
```

```
[1.28970038 1.42858145]
[0.47445426 0.52554574]
```

In [0]:
```
corpus = np.array([[0,3,0,3,0,3,0,3,0,3,0,3], [0,2,0,2,0,2,0,2,0,2,0,2,0], [1,2,1,2,1,2
hmm = HMM(num_states=2, num_words=4)

total_logprob = 0
for i, review in enumerate(corpus):
    logprob, log_xi, log_gamma = hmm.forward_backward(review)
    # your code here
    total_logprob += logprob

    words_onehot = np.eye(hmm.num_words)[review]
    max_log_gamma = np.max(log_gamma)
    simplified_gamma = np.exp(log_gamma - max_log_gamma)
    simp_gamma_by_word = simplified_gamma.T @ words_onehot + hmm.eps  # add epsilon fo
    log_gamma_by_word = np.log(simp_gamma_by_word) + max_log_gamma

    if i == 0:
        expected_si = logsumexp(log_gamma[0:-1], axis=0)
        expected_sij = logsumexp(log_xi[0:-1], axis=0)
        expected_sj = logsumexp(log_gamma, axis=0)
        expected_sjwk = log_gamma_by_word
    else:
        np.logaddexp(expected_si, logsumexp(log_gamma[0:-1], axis=0), out=expected_si)
        np.logaddexp(expected_sij, logsumexp(log_xi[0:-1], axis=0), out=expected_sij)
        np.logaddexp(expected_sj, logsumexp(log_gamma, axis=0), out=expected_sj)

        np.logaddexp(expected_sjwk, log_gamma_by_word, out=expected_sjwk)

wanted_si = np.zeros((hmm.num_states, ))
```

```python
wanted_sij = np.zeros((hmm.num_states, hmm.num_states))  # E(si -> sj): shape (num_sta
wanted_sj = np.zeros((hmm.num_states,))  # E(sj): shape (num_states,)
wanted_sjwk = np.zeros((hmm.num_states, hmm.num_words))  # E(sj,wk): shape (num_states
for idx, review in enumerate(corpus):
    log_alpha = hmm.forward(review)
    alpha = np.exp(log_alpha)
    logprob_forward = logsumexp(log_alpha[len(review)-1, :])

    log_beta = hmm.backward(review)
    beta = np.exp(log_beta)

    logprob, log_xi, log_gamma = hmm.forward_backward(review)
    xi = np.exp(log_xi)
    gamma = np.exp(log_gamma)

    for i in range(hmm.num_states):
        for t in range(len(review)-1):
            wanted_si[i] += gamma[t,i]

    for i in range(hmm.num_states):
        for j in range(hmm.num_states):
            for t in range(len(review)-1):
                wanted_sij[i,j] += xi[t,i,j]

    for j in range(hmm.num_states):
        for t in range(len(review)):
            wanted_sj[j] += gamma[t,j]

    # if idx != 0: continue
    # print(gamma)
    # print(review)
    for j in range(hmm.num_states):
        for w in range(hmm.num_words):
            for t in range(len(review)):
                if review[t] == w:
                    wanted_sjwk[j, w] += gamma[t,j]

for i in range(hmm.num_states):
    # print(expected_si[i], np.log(wanted_si[i]))
    # print(np.exp(expected_si[i]), wanted_si[i])
    assert np.isclose(expected_si[i], np.log(wanted_si[i]))

for i in range(hmm.num_states):
    for j in range(hmm.num_states):
        assert np.isclose(expected_sij[i,j], np.log(wanted_sij[i,j]))

# print(expected_sj)
# print(np.log(wanted_sj))
```

```
        for j in range(hmm.num_states):
            assert np.isclose(expected_sj[j], np.log(wanted_sj[j]))

        # print(expected_sjwk)
        # print(np.log(wanted_sjwk))
        for i in range(hmm.num_states):
            for w in range(hmm.num_words):
                assert np.isclose(expected_sjwk[i,w], np.log(wanted_sjwk[i,w]))

        # np.logaddexp(expected_sij, logsumexp(log_xi[0:-1], axis=0), out=expected_sij)
        # np.logaddexp(expected_sj, logsumexp(log_gamma, axis=0), out=expected_sj)

        # words_onehot = np.eye(self.num_words)[review]
        # max_log_gamma = np.max(log_gamma)
        # simplified_gamma = np.exp(log_gamma - max_log_gamma)
        # simp_gamma_by_word = simplified_gamma.T @ words_onehot + self.eps  # add epsilon for
        # log_gamma_by_word = np.log(simp_gamma_by_word) + max_log_gamma

        # np.logaddexp(expected_sjwk, log_gamma_by_word, out=expected_sjwk)
```

0.4 Experiments

Train a model:

```
In [318]: tokenizer = lab_util.Tokenizer()
          tokenizer.fit(train_reviews)
          train_reviews_tk = tokenizer.tokenize(train_reviews)
          print(tokenizer.vocab_size)

          hmm = HMM(num_states=10, num_words=tokenizer.vocab_size)
          hmm.learn_unsupervised(train_reviews_tk, 10)
```

```
2006
log-likelihood -2089917.1991944504
log-likelihood -1524887.1304268788
log-likelihood -1524185.523647732
log-likelihood -1523412.772067245
log-likelihood -1522495.8381112337
log-likelihood -1521350.8320171814
log-likelihood -1519871.4118507395
log-likelihood -1517913.31245186
log-likelihood -1515272.647603099
log-likelihood -1511661.0869601287
```

Let's look at some of the words associated with each hidden state:

```
In [275]: for i in range(hmm.num_states):
              most_probable = np.argsort(hmm.B[i, :])[::-1][:10]
```

```
            print(f"state {i}")
            for o in most_probable:
                print(tokenizer.token_to_word[o], hmm.B[i, o])
            print()
```

```
state 0
. 0.09441054242543682
i 0.07375932838554647
<unk> 0.035941882391585966
it 0.03331877851197925
, 0.03106381307967656
they 0.03023826336974981
that 0.022137165229001256
and 0.020421180373083766
not 0.019489947629475077
you 0.014195313828903622

state 1
br 0.12176408273555041
, 0.09291446561364468
<unk> 0.09034711996799845
. 0.0659456278307458
the 0.033882367610778695
but 0.024677347973385864
i 0.021000502852190524
this 0.01950368593574959
is 0.019205750882931572
it 0.018451345223994327

state 2
. 0.08378024253532976
the 0.08316872013124443
<unk> 0.06267266463746844
i 0.04092554562746479
to 0.02792907382439079
of 0.02121551827847617
and 0.020749482167022375
this 0.017798106510406583
, 0.01753363453933246
my 0.015819512005304567

state 3
a 0.06877201431996974
<unk> 0.054663486178699
and 0.053593120003017795
the 0.05108793288252669
to 0.04970410656083046
. 0.038678973627255274
```

```
, 0.03763810530416528
not 0.034643715563881336
in 0.02224082604480798
little 0.013879539547936607

state 4
<unk> 0.13149855982052133
. 0.03951301015366121
a 0.029893492237588772
the 0.02807700850888984
like 0.018056999785343253
in 0.017749000235563108
and 0.01517227919892736
but 0.01344810535555898
was 0.012246967951761917
for 0.011682937377019264

state 5
. 0.10667177103825425
<unk> 0.06465331901051229
i 0.05643537276107894
the 0.04081942118781156
is 0.034156494440733376
a 0.023385125891950808
and 0.022010645483575623
are 0.020154187103142034
was 0.02015102416965205
this 0.018204334497458795

state 6
<unk> 0.11426570228511333
the 0.051663134016225847
of 0.03323590458022701б
it 0.028984588403912142
to 0.027829320419171282
this 0.02267960462618376
my 0.022616415436390824
and 0.018686126427179088
for 0.014581281233718126
them 0.012525630367102922

state 7
<unk> 0.1287932321469229
. 0.07381385411044272
the 0.06182351637889356
and 0.0489805037917137
, 0.0405169611089532
in 0.02483564751060873
```

```
it 0.019576604387935674
that 0.012979725214314786
i 0.012120338900106125
just 0.011333972990936265

state 8
i 0.08384585719339639
a 0.056264733361288295
, 0.05554737280765681
have 0.030017850935334787
and 0.025733676322805956
the 0.021309160630984023
is 0.019070421754305494
you 0.01872634839399412
. 0.017973985634848526
! 0.013661761205371324

state 9
. 0.09897079245710175
<unk> 0.07636008119633064
of 0.07614743306791881
, 0.06818750043155516
to 0.04269985606081056
it 0.03294419709213846
a 0.030849564461663002
for 0.025972426870009266
on 0.020302482647297247
and 0.01767813874185242
```

We can also look at some samples from the model!

```
In [320]: for i in range(10):
              print(tokenizer.de_tokenize([hmm.generate(10)]))
```

```
['during <unk> i up all , br i in single']
['just <unk> baby bag for salty ! tastes want go']
['cats plus away i get <unk> show found i stick']
['form <unk> <unk> when yes i times would little and']
['snack salt changed saw not , ! from have spent']
['<unk> the choices and a <unk> company or easily do']
['spread good healthier being but pack com my had just']
['<unk> less , . around ! <unk> much taste other']
['i bag them immediately protein my you taste . sodas']
['along if but pancake would the , some healthier unpleasant']
```

Finally, let's repeat the classification experiment from Parts 1 and 2, using the *vector of expected hidden state counts* as a sentence representation.

(Warning! results may not be the same as in earlier versions of this experiment.)

```
In [321]: def train_model(xs_featurized, ys):
            import sklearn.linear_model
            model = sklearn.linear_model.LogisticRegression()
            model.fit(xs_featurized, ys)
            return model

          def eval_model(model, xs_featurized, ys, verbose=True):
            pred_ys = model.predict(xs_featurized)
            if verbose: print("test accuracy", np.mean(pred_ys == ys))
            return np.mean(pred_ys == ys)

          def training_experiment(name, featurizer, n_train, verbose=True):
              if verbose: print(f"{name} features, {n_train} examples")
              train_xs = np.array([
                  hmm_featurizer(tokenizer.tokenize([review]))
                  for review in train_reviews[:n_train]
              ])
              train_ys = train_labels[:n_train]
              test_xs = np.array([
                  hmm_featurizer(tokenizer.tokenize([review]))
                  for review in test_reviews
              ])
              test_ys = test_labels
              model = train_model(train_xs, train_ys)
              acc = eval_model(model, test_xs, test_ys)
              if verbose: print()
              return acc

          def hmm_featurizer(review):
              review = review[0]
              _, _, gamma = hmm.forward_backward(review)
              return gamma.sum(axis=0)

          training_experiment("hmm", hmm_featurizer, n_train=100)
```

```
hmm features, 100 examples
test accuracy 0.508
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarni
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
```

```
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

Out[321]: 0.508

Part 3: Lab writeup

1. What do the learned hidden states seem to encode when you run unsupervised HMM training with only 2 states? What about 10? What about 100?

2. As before, what's the relationship between # of labeled examples and usefulness of HMM-based sentence representations? Are these results generally better or worse than in Parts 1 and 2 of the homework? Why or why not might HMM state distributions be sensible sentence representations?

In [322]:
```python
# To find lower limit for logistic regression convergence
train_sizes = [200, 300, 500, 750, 1000, 1500, 2000, 2500, 3000]

for n in train_sizes:
    training_experiment("hmm", hmm_featurizer, n_train=n)
```

```
hmm features, 200 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.574

hmm features, 300 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

```
test accuracy 0.642

hmm features, 500 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.602

hmm features, 750 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.606

hmm features, 1000 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.612

hmm features, 1500 examples
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.614

hmm features, 2000 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.618

hmm features, 2500 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.616

hmm features, 3000 examples
test accuracy 0.616



/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.
```

```
Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

In [323]:
```python
# jk looks like all are bad for #states=10
train_sizes = [10, 20, 30, 50, 75, 100, 200, 300, 500, 750, 1000, 1500, 2000, 2500, 3

hmm_10_results = []

for n in train_sizes:
    hmm_10_results.append(training_experiment("hmm", hmm_featurizer, n_train=n, verbo
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.468


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.45


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
```

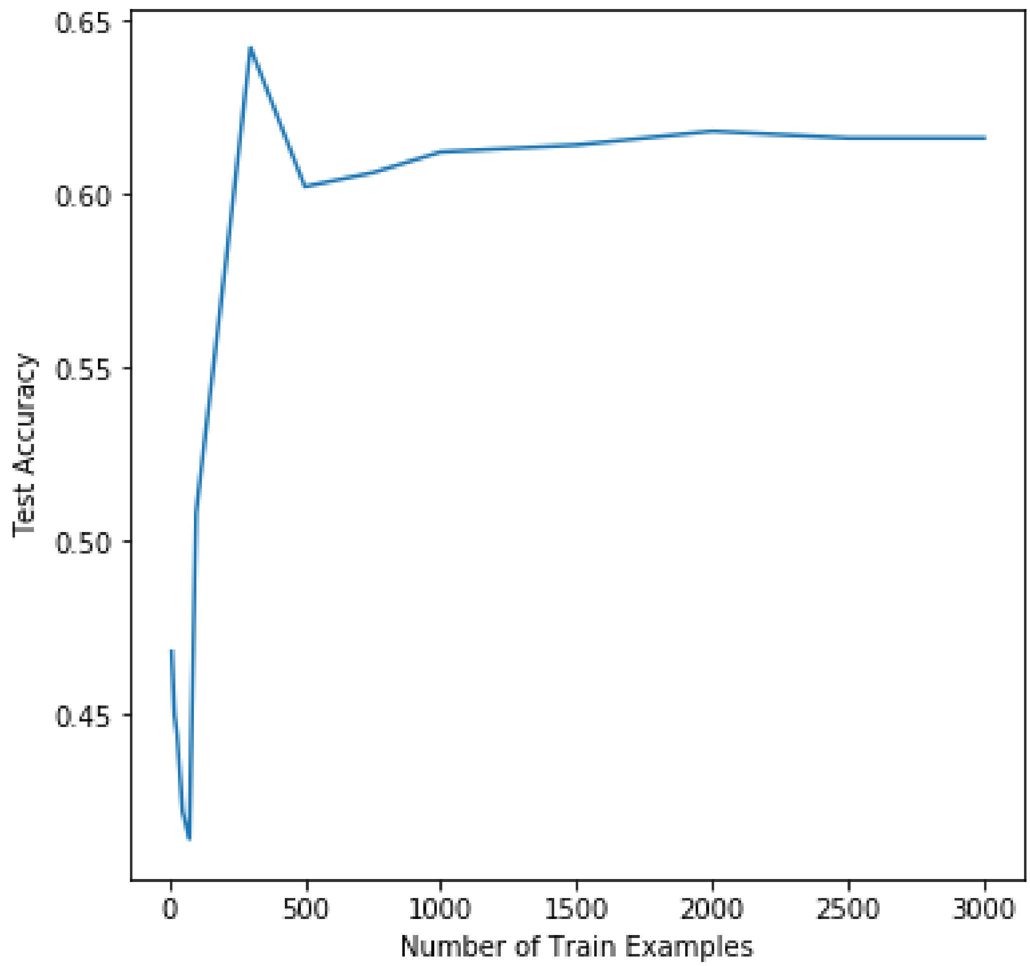
```
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.444


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.422


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.414


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.508


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
```

```
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.574


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.642


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.602


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.606


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.
```

```
Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.612


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.614


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.618


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.616
test accuracy 0.616
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

```
In [330]: plt.figure(figsize=(6, 6))
          plt.plot(train_sizes, hmm_10_results, label="hmm")
          # plt.legend(loc = "lower right")
          # plt.title('Test Accuracy of HMM-based Representations with 10 Hidden States on Dif
          plt.xlabel('Number of Train Examples')
          plt.ylabel('Test Accuracy')

          # plt.savefig('hmm_ntrain.png')
          # files.download('hmm_ntrain.png')
          plt.show()
```



In [14]:
```
tokenizer = lab_util.Tokenizer()
tokenizer.fit(train_reviews)
train_reviews_tk = tokenizer.tokenize(train_reviews)
print(tokenizer.vocab_size)

hmm2 = HMM(num_states=2, num_words=tokenizer.vocab_size)
hmm2.learn_unsupervised(train_reviews_tk, 10)

for i in range(hmm2.num_states):
    most_probable_2 = np.argsort(hmm2.B[i, :])[::-1][:10]
    print(f"state {i}")
    for o in most_probable_2:
        print(tokenizer.token_to_word[o], hmm2.B[i, o])
    print()
```

```
2006
log-likelihood -2091938.7828791158
log-likelihood -1525759.6139374922
log-likelihood -1525452.544514095
log-likelihood -1525196.4582913124
log-likelihood -1524963.8515883384
log-likelihood -1524736.7491449362
log-likelihood -1524502.0284070147
log-likelihood -1524249.7522469894
log-likelihood -1523972.7658050563
log-likelihood -1523666.0009118551
state 0
<unk> 0.10173966058336177
and 0.043608580042030824
the 0.029500988164168577
i 0.028734435183907198
it 0.028385692301300905
to 0.02548693917115605
of 0.02425388747515287
. 0.019874445558347602
br 0.0189519945042716
for 0.013088892881360028

state 1
. 0.11489358919612605
, 0.06314141567349009
<unk> 0.050067477457172715
the 0.04707051540179577
a 0.034755652784691886
i 0.030292078886628146
```

```
is 0.021600135446078216
this 0.017817989953622652
that 0.013896192304025948
as 0.010761163716723036
```

```
In [15]: for i in range(10):
             print(tokenizer.de_tokenize([hmm.generate(10)]))
```

```
['not <unk> <unk> are are <unk> are not are <unk>']
['are are <unk> these <unk> not not are are are']
['not are not <unk> <unk> <unk> not are not <unk>']
['are not <unk> are are not not are <unk> these']
['<unk> <unk> <unk> <unk> <unk> <unk> <unk> <unk> not not']
['<unk> not not are <unk> <unk> not are not not']
['<unk> not are <unk> not not not <unk> <unk> not']
['<unk> <unk> <unk> <unk> not <unk> these <unk> <unk> are']
['are not <unk> not not not <unk> are <unk> not']
['not not these <unk> <unk> <unk> these <unk> not <unk>']
```

```
In [16]: tokenizer50 = lab_util.Tokenizer()
         tokenizer50.fit(train_reviews)
         train_reviews_tk = tokenizer50.tokenize(train_reviews)
         print(tokenizer50.vocab_size)

         hmm50 = HMM(num_states=50, num_words=tokenizer50.vocab_size)
         hmm50.learn_unsupervised(train_reviews_tk, 10)
```

```
2006
log-likelihood -2078086.4314356584
log-likelihood -1525345.931128712
log-likelihood -1525212.8916231303
log-likelihood -1525077.4076970222
log-likelihood -1524935.6948303515
log-likelihood -1524783.3620497612
log-likelihood -1524614.9498223907
log-likelihood -1524423.254373716
log-likelihood -1524198.2785475415
log-likelihood -1523925.5468200485
```

```
In [23]: for i in range(hmm50.num_states):
             most_probable = np.argsort(hmm50.B[i, :])[::-1][:10]
             print(f"state {i}")
             for o in most_probable:
                 print(tokenizer50.token_to_word[o], hmm50.B[i, o])
             print()
```

```
state 0
. 0.13357598539401402
and 0.04970710984142213
to 0.03713581602758531
i 0.034144943706297386
it 0.03213097577305458
<unk> 0.020755137834818856
a 0.015988097084972275
, 0.015592170407548204
the 0.015038233535990209
in 0.012332338550948951

state 1
the 0.07098796264524891
<unk> 0.05716581508292614
. 0.048512197025335664
a 0.04809956021703975
and 0.03629223329029702
it 0.0353653557046032
, 0.02844681053321185
is 0.01922100994796981
in 0.016030475683709956
to 0.015323226765717315

state 2
the 0.06906881457874227
, 0.06146344152279113
<unk> 0.04452558688897632
it 0.03938063072879583
. 0.034586832626620316
and 0.026818188668161786
for 0.021916794594798044
in 0.02032497462957678
to 0.018473802180453323
with 0.013388953633070222

state 3
i 0.09197383181700625
<unk> 0.08941947287110451
, 0.05365318935811315
. 0.038321170103563514
of 0.03149647371717822
a 0.0173919461158527
these 0.016526935189500446
this 0.016473758166895586
and 0.012964459883268383
br 0.012913661404696533
```

```
state 4
<unk> 0.0898155710239038
. 0.063231137636617
and 0.05117748314288154
a 0.047525314526488935
to 0.03358549267849003
in 0.026387716294359126
is 0.026096207148262968
of 0.01767035548612207
was 0.01609180413102257
not 0.015711052484514505

state 5
. 0.07527909100696237
, 0.04508528282744235
<unk> 0.04249041117293242
of 0.03782986623639959
and 0.036167658441353674
i 0.03267950172913366
is 0.026981443043418138
a 0.026114141207581782
it 0.023768074506057037
in 0.017595230420778273

state 6
i 0.09440235472092204
<unk> 0.054054762566400946
the 0.04786765094107878
a 0.04378715776602611
. 0.0428776660262734
and 0.03894268309018315
br 0.0338311803739358
it 0.02843894065131895
my 0.013655680697088645
not 0.01330463053509941

state 7
, 0.05583903556391959
and 0.03977887021023833
in 0.0302534294640414
to 0.02788122734785859
for 0.025025994947814566
the 0.023147366684598884
i 0.019708902167929577
my 0.01823133899684085
<unk> 0.017766016757181938
a 0.017592573373432296
```

```
state 8
i 0.07221485320978387
. 0.06541158378714487
the 0.046638881601238746
<unk> 0.03841302020193987
this 0.03549505004468274
to 0.027182581106166902
, 0.02500946953127829
it 0.01942305087024161
not 0.013512349164053779
and 0.012998651220261457

state 9
, 0.05231726835856042
<unk> 0.05030579715102906
the 0.045470643557201866
a 0.037444522936492086
. 0.03346842648897892
and 0.030779588811831297
it 0.029599025029012283
to 0.02028056109449371
i 0.019053400732132516
not 0.01720477211231885

state 10
<unk> 0.13043645678400295
. 0.07975871510987152
, 0.055488902493307776
the 0.03284349774092562
br 0.028405102787253708
i 0.024389062977084295
and 0.021165363776250463
is 0.01999822979836433
that 0.013717517330436147
they 0.01302326725611485

state 11
<unk> 0.12210230014380977
. 0.09929089765291434
, 0.05256935271506675
i 0.03023415140925048
and 0.02457276751977981
of 0.024534133339863186
that 0.014747553325573934
in 0.012996272162790791
for 0.012266680323902388
to 0.011729512458386214
```

```
state 12
<unk> 0.19131793790929746
the 0.048880551436851695
this 0.04239459878564892
i 0.03944813859386762
it 0.03195576651559617
of 0.02152590522949218
to 0.02054906198253551
a 0.01569347188647533
is 0.012199726163016303
are 0.011439417426358988

state 13
. 0.11042685294937572
, 0.06606814591019708
to 0.04168872717974999
<unk> 0.03861452887883402
br 0.0380035398962323
the 0.027805297082229903
it 0.02206608971959026
i 0.019471758390200704
a 0.017789496380286176
with 0.015851692258264383

state 14
<unk> 0.08950648315147894
, 0.07833396022143799
the 0.041013082723846794
and 0.03948550093787296
to 0.03889567574196114
a 0.027278323844947303
of 0.02448901758477
this 0.020022812360009124
br 0.018331703679673524
. 0.013764322131727357

state 15
<unk> 0.1054070844459089
, 0.06882536238413492
the 0.0382553438566142
to 0.029017578540765802
and 0.02901112125893665
is 0.02002735856655722
but 0.01990496995461105
! 0.017091941262126355
it 0.014809312317800482
not 0.014801014531107949
```

```
state 16
<unk> 0.10759925850060434
the 0.07501489057796816
. 0.0732859123184837
a 0.04366701094606771
it 0.03368410026458485
, 0.02731001642525638
in 0.022409633914739605
is 0.021870252644507315
and 0.015139800549119224
that 0.013579781917550222

state 17
, 0.06744892573629273
of 0.03961678879189001
the 0.03579906335484142
a 0.032086340368718796
i 0.027674927265122037
it 0.021093261271356228
is 0.020223113762470275
. 0.019417837931941025
for 0.018011445621546794
you 0.01644009472459928

state 18
. 0.07708782443074592
the 0.06508154897721052
i 0.04448449805587544
a 0.042951074905443136
it 0.03374762812257114
of 0.030172167592315155
is 0.028404063157722995
this 0.025250249490764157
br 0.016677940470357658
, 0.015390791601715203

state 19
. 0.10419999577701097
<unk> 0.09978984415493886
a 0.038555935079954806
, 0.03719850626374941
of 0.03162537688689429
it 0.029491451243302297
i 0.015756465599649237
is 0.014197337641339104
in 0.0133645713916965
this 0.013116156257430112
```

state 20
, 0.05442213424244397
. 0.04033529919178106
to 0.03702746053386291
br 0.023368082986479216
are 0.022402721158774452
is 0.02217746164690812
<unk> 0.021586981492733342
in 0.02043616862653968
have 0.018252987328058915
not 0.014376359254559925

state 21
<unk> 0.13264741157163953
. 0.08888529111251352
the 0.05280898509840406
, 0.038292847856927215
of 0.0334109560501877
a 0.03240842915137756
for 0.016998864808399506
it 0.015417452867261067
is 0.014196219617548392
and 0.012985752445812895

state 22
<unk> 0.1957512114038947
. 0.08461956506962792
the 0.06031560767556544
and 0.027685090357351817
, 0.0236884625543751
of 0.021070600444743817
to 0.01317409065030281
for 0.012235589390020908
it 0.011385859320763914
that 0.009380312807952183

state 23
the 0.06820021557200823
<unk> 0.06473884978020801
and 0.05329697396399862
. 0.04231361108024892
to 0.03993295965355895
is 0.023602702454686045
of 0.016315043590147675
a 0.01219085838422458
, 0.011793892947370736
not 0.010829300856542682

state 24
. 0.1289069009617367
i 0.047699209552439205
the 0.04434461053611091
<unk> 0.03887599080122308
br 0.03577511130105288
is 0.025704679199486775
, 0.019463103669911202
with 0.014761472540175437
to 0.014130584130276777
this 0.01338943485040459

state 25
i 0.09497507348099385
the 0.054591940136402735
<unk> 0.04959744355680217
it 0.036311673778699614
and 0.03375985412567949
. 0.0319277309704274
to 0.023229728669904175
, 0.02311942768725358
a 0.021763530557924383
br 0.020392351327953116

state 26
. 0.08616402705386458
, 0.03925266964564496
and 0.03317352760611349
this 0.030229497452084163
is 0.02662299295479036
<unk> 0.02554698760739387
the 0.024025718963352538
a 0.02364871023708196
i 0.02196797036798765
to 0.019702800266698066

state 27
<unk> 0.11821259038798303
the 0.048325649506538684
, 0.03206989856201883
of 0.03139725877841939
it 0.02880078820185752
to 0.02313070090007937
is 0.022247831782450378
a 0.021880831694300645
. 0.01834874106234143
i 0.016669505638551088

state 28
. 0.11568976210262111
the 0.06750063614432902
<unk> 0.054112966516865875
br 0.03300827290856748
a 0.031871675228422935
to 0.02565209149673838
of 0.019712762729358974
and 0.012711774875459521
that 0.012669522888183935
is 0.012204477670145647

state 29
<unk> 0.1553569073906132
, 0.05302397467001936
the 0.049617835966270854
this 0.02428035879621469
and 0.021893356237657213
is 0.021778330058086267
of 0.015443884132705825
to 0.01467297059399048
in 0.014594843145687127
a 0.014583369305050657

state 30
<unk> 0.11412534317028884
, 0.07712128510221529
. 0.061730119168866504
the 0.03159445091234894
it 0.03067332289115271
to 0.01811632604461964
in 0.01321450974712446
that 0.013141386576419195
is 0.013017604905587161
for 0.012541789560830567

state 31
. 0.0903619371563553
<unk> 0.07250536701282351
, 0.05671738893093779
of 0.030592941955571114
a 0.02574071774423671
the 0.023683960732158137
to 0.020321351786856345
and 0.012991631661520767
for 0.012242902688169858
br 0.010265203181707011

```
state 32
<unk> 0.08653777051454081
. 0.05742591414500421
i 0.05337963852062644
the 0.0459613529713838
, 0.04268543917758257
and 0.026699365571669825
to 0.026660268255970877
is 0.021389710852502793
a 0.019799088056100807
br 0.015274240387978666

state 33
i 0.04567635188635884
. 0.03808481636206712
a 0.03678157487721447
, 0.03552152376696268
the 0.03239144489623098
and 0.03203370026354512
to 0.026711670280189965
this 0.026447005093795124
<unk> 0.024539604142694068
in 0.020517933491694968

state 34
<unk> 0.1097142121766752
i 0.05706507859917726
, 0.052885782256799144
. 0.05200332456594608
and 0.03819442020895518
br 0.03766636389088792
a 0.023847846615785172
it 0.019397818763901283
of 0.018791548039613023
in 0.018305961629649117

state 35
. 0.07804044468723975
<unk> 0.06135110171373102
br 0.03680502321040091
this 0.021101401294690035
to 0.01850753456612111
for 0.017809733220752647
but 0.017026094437054915
i 0.01668810060562197
they 0.01579640052052721
and 0.015593758070113236
```

```
state 36
. 0.11677525603489004
<unk> 0.10221812897959065
, 0.045304551871304576
and 0.042394706052097254
i 0.03895279890953993
the 0.03074585669387322
it 0.023380628228817035
in 0.01816491717519267
a 0.013966379164355327
of 0.013790356295237335

state 37
. 0.07553658448203387
the 0.054946192687985194
a 0.05073589491111157
i 0.024706508660935173
it 0.0233220522372 1177
br 0.020699223578338297
you 0.015881848378360233
this 0.01585982524210169
was 0.014824182602878551
but 0.01339719715572706

state 38
. 0.11155902541254276
a 0.0345447891036453
the 0.029377800667401796
it 0.02552629635228693
for 0.024287994473825576
this 0.023556097001408167
, 0.02258174625838138
and 0.021871313089426376
i 0.0203937185263239
in 0.01579851999632675

state 39
the 0.08111417936228824
. 0.07219908119386034
<unk> 0.04891379714542026
i 0.04225628436467083
and 0.037588092204266926
, 0.025826442484445253
it 0.02400747684258407
of 0.020214512596756027
this 0.019082784163968674
a 0.015552932874820017
```

```
state 40
<unk> 0.10692584502115975
. 0.09792533799466491
i 0.046742362014538755
and 0.029869177143390078
to 0.025219468436195873
br 0.024555782596663904
of 0.023596886033838204
the 0.02321477207248706
, 0.02165523360519356
a 0.014532811445546544

state 41
<unk> 0.12011927295606283
the 0.053377947009396486
and 0.052998689592164194
a 0.04312928949959183
i 0.02393037137339588
for 0.01733310036156482
you 0.014553057184064022
to 0.014208776137815128
is 0.013647769343789445
with 0.012958873226863147

state 42
a 0.04714371879655559
it 0.03378223893305346
i 0.03218591625123606
to 0.02562750574088384
, 0.025398538260956293
the 0.022079419524960314
my 0.01571678598985893
<unk> 0.015351187826544923
! 0.01505785171341468
and 0.014826547260633505

state 43
<unk> 0.1179951415532188
i 0.04241191213077853
the 0.030058611959688
, 0.028674109672256393
a 0.025860513415047692
and 0.01822559151423301
these 0.01756518141928197
for 0.015367429903409661
my 0.013714559820496745
of 0.013307340900385368
```

```
state 44
. 0.0915840890287296
, 0.05802258505148096
the 0.04666994876456948
and 0.04111529058217679
i 0.04106993006149287
to 0.02054635330528549
is 0.020400740905947696
but 0.01617251226876323
<unk> 0.014107529534377987
in 0.013664647982415581

state 45
<unk> 0.1352728334665721
. 0.07179126390103999
i 0.07163962264554104
it 0.02783102708218768
a 0.023120022215141915
br 0.022945024768671395
to 0.02036097190938408
the 0.015523118182952086
not 0.012812339355579737
was 0.012519214591613355

state 46
<unk> 0.14198024384734598
. 0.07782283415126974
to 0.03426033008706623
and 0.031027392263882937
i 0.027949617517757472
is 0.024981175085654558
a 0.02479197976264268
the 0.020852357657639792
of 0.01925252076115371
br 0.01686552844619547

state 47
. 0.08651098661378936
i 0.04582816294478209
<unk> 0.04466248554124663
this 0.0415793638070705
a 0.03205124030022724
the 0.026357140526513438
to 0.023426323335232792
is 0.02335558952822949
of 0.021360017281273617
for 0.019162439954678593
```

```
state 48
. 0.09970877260489569
i 0.05482159205915553
<unk> 0.051453728534145454
the 0.024692872578040936
br 0.015583309113155357
of 0.015212970392467402
, 0.01424257132609748
a 0.012574438494118007
have 0.012077333424432898
my 0.011319860678876296

state 49
<unk> 0.16664920112788995
the 0.04982552058309522
and 0.030792501436700766
a 0.03032793265623367
, 0.020406390774689274
it 0.01835401985868985
this 0.0182912307840841
i 0.017132530466109075
. 0.01704636768370969
br 0.014344381608555175
```

```
In [25]: for i in range(10):
             print(tokenizer50.de_tokenize([hmm50.generate(10)]))
```

```
['allergic a it br over you purchase only about for']
['it what one in have chips if <unk> in is']
["the tastes flowers during when but to and doesn't more"]
['just . this still if i are eaten the i']
['with , . br calories high over me when toddler']
['crunchy powder since school br again to their of i']
['perfect pack . and 10 <unk> , brands high it']
['come eat amazon other am have . i much i']
['. product i contained <unk> ! looking but coffee away']
['packaging is it caramel actually br br with these how']
```

```
In [327]: train_sizes = [10, 20, 30, 50, 75, 100, 200, 300, 500, 750, 1000, 1500, 2000, 2500, 3

          hmm_50_results = []

          def hmm50_featurizer(review):
              review = review[0]
              _, _, gamma = hmm50.forward_backward(review)
```

```
            return gamma.sum(axis=0)

        for n in train_sizes:
            hmm_50_results.append(training_experiment("hmm", hmm50_featurizer, n_train=n))
```

```
hmm features, 10 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.468

hmm features, 20 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.45

hmm features, 30 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.444
```

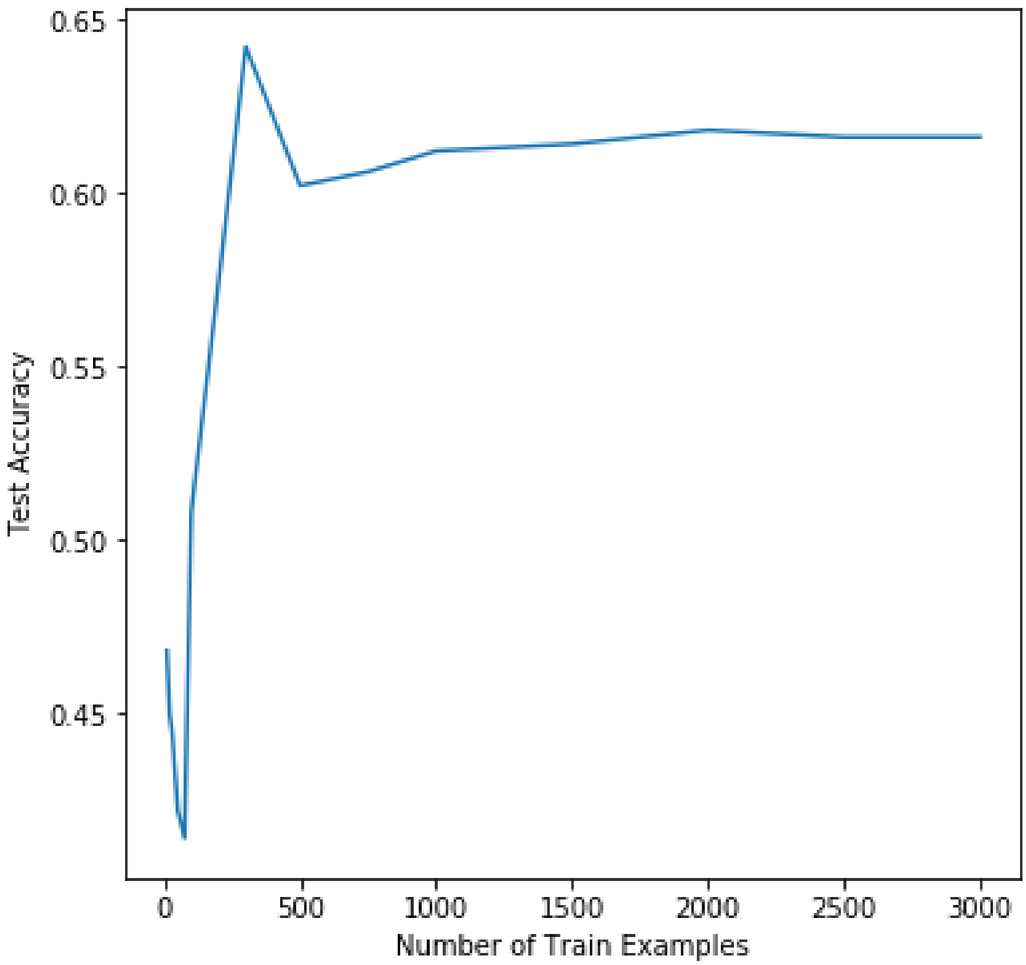
```
hmm features, 50 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.422

hmm features, 75 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.414

hmm features, 100 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.508

hmm features, 200 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.
```

```
Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.574

hmm features, 300 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.642

hmm features, 500 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.602

hmm features, 750 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
```

```
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.606

hmm features, 1000 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.612

hmm features, 1500 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.614

hmm features, 2000 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

```
test accuracy 0.618

hmm features, 2500 examples


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


test accuracy 0.616

hmm features, 3000 examples
test accuracy 0.616



/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

```
In [331]: from matplotlib import pyplot as plt

          plt.figure(figsize=(6, 6))
          plt.plot(train_sizes, hmm_50_results, label="hmm")
          # plt.legend(loc = "lower right")
          # plt.title('Test Accuracy of HMM-based Representations with 50 Hidden States on Dif
          plt.xlabel('Number of Train Examples')
          plt.ylabel('Test Accuracy')

          # plt.savefig('hmm_ntrain.png')
          # files.download('hmm_ntrain.png')
          plt.show()
```



In [329]:
```
tokenizer = lab_util.Tokenizer()
tokenizer.fit(train_reviews)
train_reviews_tk = tokenizer.tokenize(train_reviews)
print(tokenizer.vocab_size)

hmm100 = HMM(num_states=100, num_words=tokenizer.vocab_size)
hmm100.learn_unsupervised(train_reviews_tk, 10)
```

```
2006
log-likelihood -2073205.3254522625
```

```
---------------------------------------------------------------------------

KeyboardInterrupt                         Traceback (most recent call last)
```

```
<ipython-input-329-63586b40b450> in <module>()
      5
      6 hmm100 = HMM(num_states=100, num_words=tokenizer.vocab_size)
----> 7 hmm100.learn_unsupervised(train_reviews_tk, 10)


    <ipython-input-315-d8b6308e364f> in learn_unsupervised(self, corpus, num_iters, verbose
    166                 else:
    167                     np.logaddexp(expected_si, logsumexp(log_gamma[0:-1], axis=0), o
--> 168                     np.logaddexp(expected_sij, logsumexp(log_xi[0:-1], axis=0), out
    169                     np.logaddexp(expected_sj, logsumexp(log_gamma, axis=0), out=exp
    170


    /usr/local/lib/python3.6/dist-packages/scipy/special/_logsumexp.py in logsumexp(a, axis
    110         tmp = b * np.exp(a - a_max)
    111     else:
--> 112         tmp = np.exp(a - a_max)
    113
    114     # suppress warnings about log of zero


    KeyboardInterrupt:
```