NOTE: some of the numerical/lexical results discussed below may slightly differ from the the attached notebook images – this is due to me re-training the models prior to converting the notebook to a PDF. I have concatenated my Notebook PDFs to the end of this report for additional reference if needed.

1 Part 1 - Matrix Factorization

1.1 *Code Implementation Details*

Learn_reps_lsa(matrix, rep_size):
• I used the np.linalg package to perform Single Value Decomposition (SVD) of the given matrix, which yields U, S (the diagonal matrix), and V. Here, U is the word representation, so with the given "rep_size," I truncate the extra columns depending on this "rep_size" value and return the corresponding word rep matrix.

Transform_tfidf(matrix):
• I first initialize an empty 2-D matrix with the same shape as the provided matrix, and an array that will store the TF-IDF values. Then for each row (which corresponds to a particular word), I determine the number of documents that this word is present in. With this information, I then multiply each row from the original matrix with the corresponding TF-IDF value, and return the new transformed matrix.

Lsa_featurizer(xs):
• "Xs" has the counts of each word in different reviews, and we have previously computed the word features using LSA (reps_tfidf). We also know that each word in the vocabulary has an LSA feature vector as a row in reps_tfidf. Hence, we know that multiplying xs by reps_tfidf yields the vector representations of all reviews. We then normalize and return the result.
• The resulting accuracy for LSA featurizer is 80.4%, which is higher than the 75.6% for word featurizer.

```
word features, 3000 examples
0.784

lsa features, 3000 examples
0.804

combo features, 3000 examples
0.814
```

1.2 *Questions*

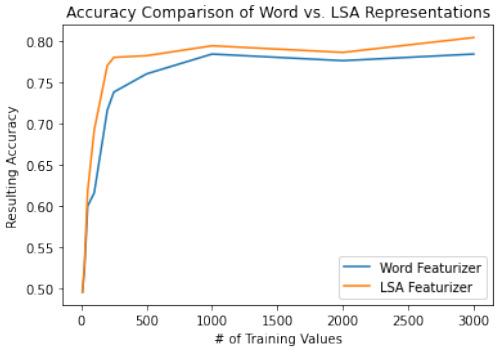
a). Using SVD, let's first represent W_{td} as $U \sum V^T$, which means that W_{td}^T is represented as $V \sum^T U^T$. To get W_{tt}, we multiply W_{td} by W_{td}^T, and we accordingly get: $(U \sum V^T) * (V \sum^T U^T) = U \sum^2 U^T$. From this, we can see that the left singular vectors of W_{td} and W_{tt} are both U – meaning that they are the same!

In the experiment I conducted within my notebook, the results yield the same result. Running "learn_reps_lsa" with my test matrix yields "U". Then multiplying the test matrix by its transpose, and using this new matrix as a new input for "learn_reps_lsa," yields "new_U." Comparing these 2 results, we easily see that the matrices are equal. The reason this occurs is that learn_reps_lsa" runs SVD on the input matrix, and as demonstrated in the proof above, the cancellation via factorization through SVD leads to the same left-singular vectors.

b). The general trend I observed from the experiment conducted within my notebook is that different rep sizes result in different results for nearest neighbors – there are changes to the representation space. For my example, let's look at the following words: "the", "dog", "3", "good". With these words, I ran "lab_util.show_similar_words()" for 3 rep sizes: 250, 500, and 1,000. The words that are most similar to the provided word set differ based on each rep size (the exact word list and ordering can be seen through running the cell). Hence, we see that there is no universal set of similar words, and increases in size of LSA representation results in both changes to the set of similar words in addition to increases in distance of the corresponding nearest neighbor words. Additionally, we can see that with a rep size that's too small we will have coarser granularity and may not fully capture some of the subtleties in the training data; with a rep size that's too high, we will overfit some of the noise in the training.

c). Learned representations do help with the review classification problem, which we can see through the experiment I developed to test this. I generated a list of different training values ([10, 25, 50, 100, 200, 250, 500, 1000, 2000, 3000]), and ran training experiments using the "word featurizer" and "lsa featurizer" for each value. This generates different accuracies, which I then plotted (as can be seen below). From this plot, we can determine different findings about the number of labeled examples and word embeddings.

We can first see that the accuracy for both representations increases as the number of training values increases. The accuracy can also be seen to reach a plateau after around 1000 training samples – for both featurizers. We then compare the accuracies corresponding to each featurizer, and it can easily be seen that the LSA featurizer consistently produces a higher training accuracy than the word featurizer throughout the entirety of the plot. This is significant, as it reaffirms our understanding in this case that learned representations are certainly effective and help in our review classification problem.



2 Part 2 - Language Modeling

2.1 *Code Implementation Details*

__Init__(self, vocab_size, embed_dim):
• We want to instantiate our model and define its layers when creating the model object. For the first layer, I chose to use an embedding layer instead of one-hot encoding layer via "nn.Embedding()". This essentially allows the model to have a layer that can store and update word embeddings and retrieve them using indices. The input parameters are:
- "Vocab_size + 1" for "num_embeddings" – we do this to account for indices of -1
- "Embed_dim" for "embedding_dim" – given size of each embedding vector
- "padding_idx = 0" – added to account for size increase of embedding dictionary

• For the model's second layer, I simply use "nn.Linear()" to apply a linear transformation to the results from "nn.Embedding()". The input parameters are:
- "embed_dim" for "in_features" - size of each input sample
- "vocab_size" for "out_features" - size of each output sample

Forward(self, context):
• Here we implement how our model is going to conduct its forward pass during training. I start by retrieving the correct word embedding corresponding to the input context.
- Embedding = self.embeddings(context)

• Then, based on the methodology of Continuous Bag of Words (CBOW), context is represented by the average of neighboring word vectors. Hence, we take the mean of the retrieved embedding representing our context.
- avg_embedding = embedding.mean(1)

• Lastly with this average embedding, we pass it through the linear classifier to produce our output, which we now return.
- output = self.linear(avg_embedding)

Learn_reps_word2vec(corpus, window_size, rep_size, n_epochs, n_batch):
• Here, I will go over certain choices I made throughout this method.
• For the loss function, I used Cross Entropy Loss (nn.CrossEntropyLoss()) – this determines the cross entropy between the predicted probabilities and the true labels. We don't need to include softmax within our forward propagation method as Cross Entropy Loss already includes it within its implementation.
• In order to account for embedding indices of -1 within a given context, I add 1 to each index so that it remains compatible with the embedding dictionary used. This allows for the prevention for index access errors,

and also matches the model's passed in "vocab_size."
• The code I implemented in the training loop follows the traditional approach for forward and backward pass, gradient update, loss calculation, and parameter update – can be seen in code.
• Lastly to return the desired embedding matrix, I extract the weights from the trained model's embeddings, remove the extra padding from the result, and convert to a numpy array.
- embedding_matrix = model.embeddings.weight.data
- return embedding_matrix[1:,:].cpu().numpy()

W2v_featurizer(xs):
• "Xs" has the counts of each word in different reviews, and we have now determined the word embedding matrix using the Word2vec-style objective (reps_word2vec). We also know that each word in the vocabulary has an average W2V feature vector as a row in reps_word2vec. Hence, we know that multiplying xs by reps_word2vec yields the average vector representation of each review. We then normalize and return the result.
• The resulting accuracy for W2V featurizer is 82%, which is higher than both the 80.4% for LSA featurizer and 75.6% for word featurizer.

```
word2vec features, 3000 examples
0.794
```

2.2 *Questions*

a). The nearest neighbors in representation space demonstrate that Word2vec does a pretty good job of identifying relatively similar words to the set of input words. Let's look at the input example of the following words: "the", "dog", "3", "good." For each of these words, it can be observed that many of the nearest neighbors for each of them maintain some level of similarity. For "the," which is an article, some of the NNs include other articles such as "their," "those," and identifying adjectives such as "another." For the word "3", all NNs are also numbers – whether that be in numerical format such as "25" or word format such as "six." We can also look at "good," which has some NNs that are synonyms including "excellent," in addition to an antonym such as "bad."

From this qualitative analysis, we can see that Word2vec performs pretty well when looking at word similarity. With this, I have realized that word similarity does not necessarily mean for NNs to be just literal synonyms for a word, but also includes words that maintain similarity with regards to structure, usage, and context.

```
3 289
  greta 1.700
  honestly 1.706
  two 1.711
  9 1.724
  four 1.725
```

b). When comparing the Word2vec and LSA approaches with the same word set ("the", "dog", "3", "good"), it can be determined that the Word2vec objective most definitely performs better and presents more "similar" nearest neighbors. Looking at the resulting NNs for each of the words, it can be seen that there is more structure to the NNs through Word2vec versus LSA. When looking at the NNs of "3", LSA struggles to produce any number values as neighbors while Word2vec easily does so. Comparing the results of "the," LSA does not classify any other articles or identifying adjectives, while Word2vec is able to do so. This can be seen through other NN findings of the other words – substantiating that Word2vec not only demonstrates a higher accuracy via its featurizer (of 82%), but also more effectively identifies similar words.

c). To better understand the effect of context size on the word representation and corresponding accuracy, I developed an experiment to test just that. Keeping all other parameters same, I trained the Word2vec model with 3 different window sizes (which is another way of representing the context sizes): 1, 2, 4. Upon generating the 3 different word embedding matrices with these context sizes, it can be determined that an increase in context size results in a general trend of increase in accuracy. For example, when running the training experiment with each of the generated word embeddings (and in my case, with 3 epochs), the accuracies for 1, 2, 4, and 8 context sizes are 0.784, 0.786, 0.792, and 0.808, respectively. Hence, the context size is indeed an important parameter for Word2vec models!

3 Part 3 - Hidden Markov Models

3.1 *Code Implementation Details*

__Init__(self, num_states, num_words):
• For self.A, we initialize a 2-D np.array with "num_states" rows and "num_states" columns, with each entry initialized as a random value between 0 and 1. Then for each row, we divide by the total sum of the row – this allows for the sum of the row to equal 1.
• For self.B, we initialize a 2-D np.array with "num_states" rows and "num_words" columns, with each entry initialized as a random value between 0 and 1. Then for each row, we divide by the total sum of the row – this allows for the sum of the row to equal 1.
• For self.pi, we initialize a vector of size "num_states", with each element initialized as a random value between 0 and 1. Then, we divide the vector by its total sum – this allows for the sum of the vector's entries to equal 1.

Forward(self, obs):
• To run the forward algorithm, we first produce log versions of the probability and distribution matrices (self.A, self.B, self.pi); we do this so we can more easily perform log calculations of the matrices.
• Then, we initialize the alpha base case values by initializing the first row of the alpha matrix with the corresponding value for each state:
- alpha[0][s] = log_pi[s] + logB[s, obs[0]] (s = state)
• Then to determine the remaining alpha values for the remaining states, we perform log based matrix operations. For this, there are a few considerations we need to make. First, we want to vectorize the matrix operations so that we eliminate the need for nested for-loops and hence reduce runtime. Additionally as we are working in the log space, we will perform modified operations instead of multiplying and summing as discussed in lecture.
• We will simply iterate over the number of observations, for which we first calculate "vectorized_alpha." "Vectorized_alpha" is the result from multiplying the log probability matrices "alpha[t-1]" and "logA" (corresponding with time step "t"), and we use the "logdot" helper function to do so, which accounts for the fact that both matrices are log representations.
• Lastly with this result, we add "vectorized alpha" with "logB[:, obs[t]]" to generate the new row of alpha values for all states (corresponding with time step "t"). Again, as we are within the log space, we add here instead of multiplying – aligning with the rules of log operations. We continue this process until completion, and return the final matrix of alpha values.

Backward(self, obs):
• To run the backward algorithm, we first produce log versions of the probability and distribution matrices (self.A, self.B, self.pi).
• Then, we initialize the beta base case values at final time step "T" by initializing the last row of the beta matrix with 0; we do this because though the value should be 1, we are in the log space, meaning that log(1) = 0.
• Then to determine the remaining beta values for the remaining states, we perform log based matrix operations. For this, there are a few considerations we need to make. First, we want to vectorize the matrix operations so that we eliminate the need for nested for-loops and hence reduce runtime. Additionally as we are working in the log space, we will perform modified operations instead of multiplying and summing as discussed in lecture.
• We will simply iterate over the number of observations, for which we first calculate "vectorized_beta." "Vectorized_beta" is the result from adding the log probability matrices "beta[t+1]" and "logB[:, obs[t + 1]]" (corresponding with time step "t"). Again, as we are within the log space, we add here instead of multiplying – aligning with the rules of log operations.
• Lastly with this result, we use the "logdot" helper function, which accounts for the fact that both matrices are log representations, to multiply "vectorized_beta" with "logA.T" and generate the new row of beta values for all states (corresponding with time step "t").
• We continue this process until completion, and return the final matrix of beta values.

Forward_backward(self, obs):
• Here, we will focus on the computation of 3 items: logprob, gamma, and xi.
• To run the forward-backward algorithm, we first produce log versions of the probability and distribution matrices (self.A, self.B, self.pi).
• For "logprob", we take the last row of alpha values (which consists of all alphas at the final time step "T), and take the "summation" of these values. However as we are dealing with the log space, we use the "logsumexp" method to do so.

• To calculate the "gamma" matrix for all states, we simply "multiply" the "alpha" matrix by the "beta" matrix, then divide by the calculated "logprob." However as we are working in the log space, we instead add the "alpha" matrix with "beta" matrix, then subtract by "logprob" (which although is a scalar, is able to automatically handle entry-wise subtraction via broadcasting). The result is the "gamma" matrix!
• For the "xi" tensor, we want to determine the marginal transition probability from each state to every other state for all timesteps. Hence, we iterate over the size of the sequences obs-1 (as we don't consider the very last timestep T). The base equation is as follows: (where the denominator is simply the previously calculated logprob value).
• Essentially to allow for vectorization, I applied the np.tile() method, in which I myself "broadcasted" the matrices that needed shape adjustment so operations can be performed on all log matrices simultaneously (the full equation in vectorized log format can be seen in the code). As we are within the log space, we add and subtract matrices instead of multiplying and dividing – aligning with the rules of log operations.
• We continue this process until completion, and return the final xi matrix of beta values.

Learn_unsupervised(self, corpus, num_iters, print_every=10):
• For the Baum-Welch algorithm, we initialize each of the following with the corresponding desired shapes: expected_si, expected_sj, expected_sij, expected_sjwk, and expected_q1 (total_logprob is simply set to 0).
• When iterating through each review in the corpus, we get the review's logprob, xi, and gamma items. We will now discuss how we calculate each of the desired expected items.
• Total_logprob is simply the summation of each review's logprob value.
• For expected_si, we can continuously add to this as long as we extract the correct summation values from gamma that correspond to expected_si's shape. With this, and the definition discussed in lecture, we can directly add "gamma[:-1, :].sum(axis=0)" – which is a vector of size (num_states,) – for each review, giving us the result.
• I defined my own expected_sj matrix, which allows for certain operations down the line to be made easier. We can continuously add to this as long as we extract the correct summation values from gamma that correspond to expected_sj's shape. With this, and the definition discussed in lecture, we can directly add "gamma.sum(axis=0)" – which is also a vector of size (num_states,) – for each review, giving us the result.
• For expected_sij, we can continuously add to this as long as we extract the correct summation values from xi that correspond to expected_sij's shape. With this, and the definition discussed in lecture, we can directly add "xi.sum(axis=0)" – which is an array of size (num_states, num_states) – for each review, giving us the result.
• For expected_q1, we simply add the first row of gamma for each subsequent review - which is a vector of size (num_states,).
• For expected_sjwk, there is a little more work that needs to be done for it to be fully determined. For a given review, we essentially iterate over the review to get the time step index and the obs value. With this, and the definition from lecture, I defined an expression that adds to the specific column in expected_sjwk that corresponds to obs at time step "t" w.r.t. our word of interest: "expected_sjwk[:, ob] += gamma[i, :].T". Taking the transpose of gamma allows for the both the shape and desired content for operation to align. Hence, iterating over each review for all reviews allows for the comprehensive determination of expected_sjwk.
• Prior to calculating A and B, we want to reshape our determined expected_si and expected_sj matrices so that they can properly operate with the appropriate dimensions of expected_sij and expected_sjwk; hence, I used ".reshape(self.num_states, 1)."
• Finally to calculate A_new, we divide expected_sij by the final expected_si; to calculate B_new, we divide expected_sjwk by the final expected_sj; and lastly for pi_new, we simply divide expected_q1 by the sum of all of its entries so that its resulting sum is equal to 1. We are now fully done with running the Baum-Walch algorithm!

```
log-likelihood -65.476923308951
After this test case, hmm.A should either be approximately, [[0 1]
 [1 0]]
This is your current value of hmm.A:  [[0. 1.]
 [1. 0.]]
After this test case, hmm.B should either be approximately, [[0.  0.  0.5 0.5]
 [0.5 0.5 0.  0. ]]  or it should be  [[0.5 0.5 0.  0. ]
 [0.  0.  0.5 0.5]]
This is your current value of hmm.B:  [[0.      0.      0.52174 0.47826]
 [0.54167 0.45833 0.      0.     ]]
```

3.2 *Questions*

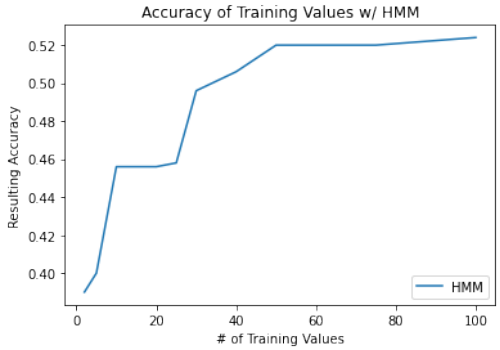
a). $A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, $B = \begin{bmatrix} 0 & 0 & 0.6 & 0.4 \\ 0.4 & 0.6 & 0 & 0 \end{bmatrix}$, $\pi = \begin{bmatrix} 1 & 0 \end{bmatrix}$

b). There seems to be different types of encodings for the different number of states during HMM training. For 2 states, we can qualitatively see that there are more verbs present than other parts of speech. In this case, this points us to the understanding that the 2 learned hidden states are more likely to observe verbs than nouns

and adjectives. For 10 states, there seems to be more distinction between the grammar. Here, we can see more unique states, such as states with specifically just nouns, verbs, articles, and adjectives. Lastly for 100 states, there is even greater "nearness" for some types of words, and many of the states exhibit even more similar probability distributions. Specifically, some words that are more frequently observed are clustered together for a specific state – meaning that they are all pretty likely observations from the state. With this, we can see that a greater number of states yields better encoding and representation of fine-grained grammar.

c). As before, as the number of labeled examples increases, the corresponding accuracy of the HMM-based representations also increase. Just as what we saw in the previous examples, there seems to be a plateau in accuracy at some point; in our example here, that seem to begin around after 50 labeled examples. There is a pretty consistent increase in accuracy over the differing number of samples up to this point, and our plot below demonstrates this positive trend.

However though the trend is similar to Parts 1 and 2, the accuracy itself seems to suffer, with it plateauing around 0.52 here, as opposed to around 0.8 for the LSA and Word2vec approaches – indicating worse performance than the earlier models. This could be attributed to the fact that HMM is working within a different space than the other two models, and the HMM uses hidden state counts to generate a feature representation. From this, the HMM just may not be best suited for these types of classification problems. But, the HMM does maintain some unique positive characteristics. The first order context present in our HMM means that it is able to pick up on certain grammatical occurs within sequence such as double negatives – something that the previous models are unable to do.



d). Bigram models essentially use the current word to help predict the following word. To implement a bigram model, let's generate an HMM with the following property: v hidden states that correspond to v words from our provided corpus. Using the training data's raw counts, we generate the initial state distribution; we can call this "pi". Then we have "B", which will contain the state observation probabilities; "B" will allow us to represent the probability of a following observation given our current state. Additionally, our HMM will follow the Markov assumption just as a traditional bigram would. With all of these components defined, we can now see that if we are in state "i" and observe word "j," we can use our determined probabilities and accordingly move to state "j". Hence, we see that the resulting state to state matrix, which we recognize as "A", acts in accordance with "B." Conclusively as we now know both "A" and "B" are of the same shape, the number of states equals the size of the vocab – giving us a v-state HMM!