Part 1. Revisiting Classification with BERT

(a) We ran the model with a couple different batch sizes and learning rates:

- Batch size of 16, learning rate of 0.00002
  `Final Accuracy: 0.876`
- Batch size of 32, learning rate of 0.00002:
  `Final Accuracy: 0.866`
- Batch size of 16, learning rate of 0.00005:
  `Final Accuracy: 0.862`
- Batch size of 32, learning rate of 0.00005:
  `Final Accuracy: 0.834`

Based on the results, a batch size of 16 and learning rate of 0.00002 produces the best performance, with a final accuracy of 0.876. This is noticeably better than the HW1 classifiers (i.e. the two BoW variants, LSA, and Word2Vec), the best of which had an accuracy of roughly 0.80 to 0.85 over a range of trials.

Part 2. Hidden Markov Models

(a) We construct an 8-state HMM with the following parameters:

$$\pi = \begin{bmatrix} 0.25 & 0.25 & 0.25 & 0.25 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A = \begin{bmatrix} 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \end{bmatrix} \qquad B = \begin{bmatrix} 1&0&0&0 \\ 1&0&0&0 \\ 0&1&0&0 \\ 0&1&0&0 \\ 0&0&1&0 \\ 0&0&0&1 \\ 0&0&1&0 \\ 0&0&0&1 \end{bmatrix}$$

Essentially, we model each sequence in the corpus as only jumping between two particular hidden states; for instance, the sequence $[0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2]$ jumps between hidden states 1 and 5 repeatedly. Then, there is a 2-to-1 mapping between hidden states and vocabulary words, since each "word" is used in two sequences and therefore corresponds to two hidden states. Lastly, we assign equal probabilities in π to each of the four hidden states representing the "start" of each sequence in the corpus.

(b) Implementation details: for our implementation, we follow the equations given in Appendix A for the Baum-Welch algorithm, but utilize matrix operations (rather than nested for loops) in order to vastly cut down on the model training time.

For 2 states, we get the following words associated with states:

```
state 0: [,, <unk>, ., and, the, br, they, but, a, or]
state 1: [<unk>, ., the, i, a, ,, to, and, it, of]
```

For 10 states, we get:

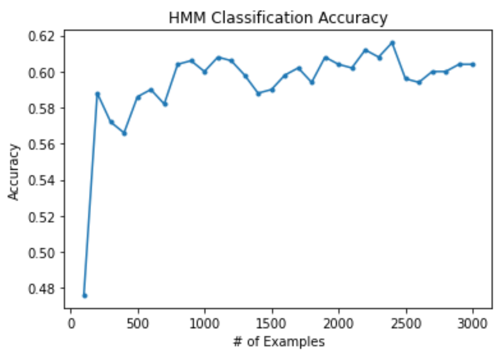
```
state 0: [<unk>, ., ,, and, a, the, of, is, are, not]
state 1: [i, <unk>, of, in, ,, to, it, br, the, we]
state 2: [<unk>, and, ., the, to, is, with, of, ,, that]
state 3: [i, this, the, my, it, these, ., if, is, it’s]
state 4: [<unk>, the, a, ., is, and, ,, you, br, not]
state 5: [<unk>, ., ,, to, of, a, i, not, taste, more]
state 6: [., <unk>, ,, to, the, it, of, br, and, but]
state 7: [., a, the, <unk>, ,, have, in, was, not, br]
state 8: [<unk>, ., they, the, a, that, are, is, i, ,]
state 9: [., <unk>, ,, and, the, it, for, br, to, !]
```

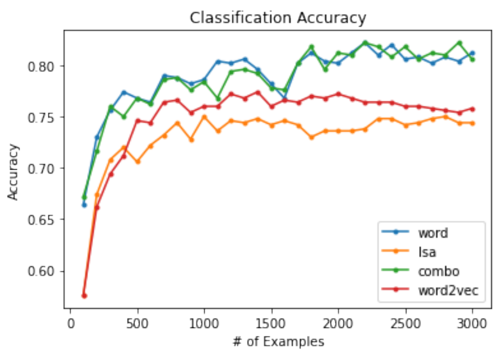
And for 100 states, we get (choosing a subset of the states):

```
state 0: [., <unk>, a, the, and, br, to, but, in, that]
state 1: [<unk>, br, to, ,, ., the, and, of, a, it]
state 2: [<unk>, ., the, of, a, is, br, in, with, that]
state 3: [., the, ,, <unk>, br, it, a, for, that, of]
state 25: [<unk>, and, it, of, a, br, i, ,, !, not]
state 50: [., ,, the, <unk>, and, of, it, is, with, i]
state 75: [., <unk>, and, the, a, to, is, i, for, them]
state 99: [<unk>, a, is, and, ,, br, ., in, was, have]
```

We see that regardless of the number of hidden states, each hidden state usually emits the most frequently occurring tokens, which are usually conjunctions (such as the, and, or, to), punctuation marks (such as , and .), or <unk>. For the HMMs with 2 or 10 hidden states, this makes sense because there are relatively fewer states to capture the most frequent tokens, so each hidden states must emit them with high probabilities. However, the fact that we see this trend continuing even for an HMM with 100 hidden states might imply that even that many hidden states is not enough to "capture" the structure of the English language, since each state is still emitting the common tokens most frequently. Thus, we may need to train an HMM with an even greater number of hidden states to begin to see some distinction between the hidden states (i.e. capturing some of the hidden patterns in the language).

(c) In HW1, we trained LSA and Word2Vec classifiers on up to 3,000 labeled examples, with accuracies roughly between 0.80 and 0.85. As seen below, the 10-state HMM model performs much worse on the same range of examples, with accuracies roughly between 0.55 and 0.62.





The LSA and Word2Vec classifiers produced nearest neighbors in the representation space for each word; for instance, for *the* the nearest neighbors for LSA were $of, ., in, on, to$; and the nearest neighbors for Word2Vec were $my, their, a, an, our$. This seems very similar to the most frequently emitted tokens for many of the states of our HMM models, indicating that some of the patterns captured by LSA are also noticed by HMM. However, since each of the states are picking up on the same "cluster" of words (namely the conjunctions such as the, to, on, etc.), the HMM state distributions might not be sensible sentence representations, since they would be repeating the same cluster of words over and over rather than emitting other words in the language, such as *good* or *dog* as in the example clusters in LSA/Word2Vec.

(d) We can implement the bigram model, given $P(w_1)$ and $P(w_t|w_{t-1})$, as a v-state HMM with the following parameters:

- π is a $1 \times v$ matrix with $\pi_i = P(w_1 = i)$
- A is a $v \times v$ matrix with $A_ij = P(w_t = j|w_{t-1} = i)$
- B is a $v \times v$ identity matrix with B_ij equal to 1 if $i = j$ and 0 otherwise.

In other words, the HMM consists of v hidden states, each representing a word in the vocabulary. The transition probabilities are simply the bigram probabilities $P(w_t|w_{t-1})$, and the initial state probabilities are the unigram first word probabilities $P(w_1)$.

Part 3. Trees

Part A

(a) Using a recursive definition for OR (object-relative clauses), we have the following CFG rules:

- `S` $\rightarrow$ `NP OR VP`
- `NP` $\rightarrow$ `D N`
- `OR` $\rightarrow$ $\emptyset$ `| C NP OR T`
- `VP` $\rightarrow$ `I`

where `N` $= \{man, dog, cat\}$, `I` $= \{meowed, barked, rant\}$, `T` $= \{feared, chased, loved\}$, `D` $= \{the\}$, and `C` $= \{that\}$.

(b) The new CFG rules are:

- `S` $\rightarrow$ `NP VP | NP OR VP | NP DOR VP`
- `NP` $\rightarrow$ `D N`
- `OR` $\rightarrow$ `C NP T`
- `DOR` $\rightarrow$ `C NP OR T`
- `VP` $\rightarrow$ `I`

Essentially, we hardcode the double-nesting limit into the CFG rules by explicitly defining OR to be a single object-relative clause and DOR to be a double object-relative clause, and making it so that a sentence can only contain an OR, a DOR, or neither.

Part B

(a) Implementation details: our SentEnc module contains an embedding layer and a bi-direction LSTM layer. We then train a classifier consisting of the SentEnc and a final linear layer, which takes a batch of word indices as input, generates the set of word embeddings using the SentEnc module, calculates the span embeddings, and outputs the logits resulting from the final linear layer.

We obtain the following results:

```
precision: 0.8490942277460672,
recall: 0.9437024355351259,
f1: 0.8939020326557814,
exact_match: 0.41647215738579524,
well_form: 0.5489607646993442,
tree_match: 0.48338335000555743,
num_examples: 8997
```

The final scores for `F1`, `exact_match`, and `tree_match` are 0.89, 0.42, and 0.48 respectively, and the percentage of well-formed predictions is 54.9%.

(b) A bi-directional LSTM has the advantage of being able to see both the past and future word sequences of a sentence, while a uni-directional LSTM can only see the past. This is advantageous for generating word and span embeddings because we are given the full text of a sentence when training the model, and therefore can fully utilize both directions of knowledge to generate embeddings, leading to more accurate tag predictions for the span classification task.

(c) Currently, we generate span embeddings by concatenating the word embeddings of the first and last word in the span. While this is quick and space-efficient, it loses out on all the information captured by the intermediary words' embeddings. To remedy this, rather than only concatenating the first and last word embeddings, we could instead concatenate all the word embeddings in the span together in order, resulting in a much longer span embedding that may result in better accuracy at the cost of efficiency.

Part C

(a) Sorry, I didn't get a chance to figure out this part before the deadline :c