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

```
Cloning into 'hw1'...
```

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

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

import lab_util
```

Hidden Markov Models

In the remaining part of the lab (containing part 3) you'll use the Baum--Welch algorithm to learn *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:

```
data = []
n_positive = 0
n_disp = 0
with open("/content/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 t
rating: 1 (good)

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

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

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

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

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

Read 4000 total reviews.
```

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

IMPORTANT NOTE: if you directly multiply probabilities as shown on the class slides, you'll get underflow errors. You'll probably want to work in the log domain (remember that `log(ab) = log(a) + log(b)`, `log(exp(a) + exp(b)) = logaddexp(a, b)`). In general, we recommend either `np.logaddexp` or `scipy.special.logsumexp` as safe ways to compute the necessary quantities.

```
import scipy.special

!pip install tqdm

import tqdm
```

```
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages (4.41.1)
```

```
# hmm model
```

```
clip = 1e-50
class HMM(object):
    def __init__(self, num_states, num_words):
        self.num_states = num_states
        self.num_words = num_words

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

        """
        Initialize the matrix A with random transition probabilities p(j|i)
        A should be a matrix of size `num_states x num_states` with rows that
        sum to 1.
        """
        self.A = np.random.uniform(size=(self.num_states, self.num_states)) # your code here
        for i in range(num_states):
          self.A[i, :] =  self.A[i, :]/np.sum(self.A[i, :])
        """
        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.uniform(size=(self.num_states, self.num_words))
        for i in range(num_states):
          self.B[i, :] =  self.B[i, :]/np.sum(self.B[i, :])

        """
        Initialize the vector pi with a random starting distribution. pi should
        be a vector of size `num_states` with entries that sum to 1.
        """
        U = np.random.uniform(size=self.num_states)
        self.pi = U/np.sum(U)

        #self.Alog = np.log(self.A)
        #self.Blog = np.log(self.B)

    def generate(self, n): # ras, very straightforward
        """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, :]
```

```
            a = self.A[state, :]
            # update the state
            state = np.random.choice(self.states, p=a)
        return sequence

    def forward(self, obs):
        """
        Runs the forward algorithm. This function should return a
        `len(obs) x  num_states` matrix where the (t, i)th entry contains
        log p(obs[:t], hidden_state_t = i)
        """
        alpha = np.zeros((len(obs), self.num_states))
        alpha[0, :] = np.log(self.pi+clip) + np.log(self.B[:, obs[0]]+clip)
        for t in range(1, len(obs)):
          alpha[t, :] = np.log(np.sum(np.exp(alpha[t-1, :] + np.log(self.A.T+clip)), axis=1)
        return alpha


    def backward(self, obs):
        """
        Run the backward algorithm. This function should return a
        `len(obs) x num_states` matrix where the (t, i)th entry contains
        log p(obs[t+1:] | hidden_state_t = i)
        """
        beta = np.zeros((len(obs), self.num_states))
        beta[:, -1] = 0
        for t in range(len(obs)-1, 0, -1):
          beta[t-1, :] = scipy.special.logsumexp(beta[t, :] + np.log(self.A+clip) + np.log(se
        return beta

    def forward_backward(self, obs):
        """
        Compute forward-backward scores

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

        gamma is a matrix of size `len(obs) x num_states1. It contains the
        marginal probability of being in state i at time t

        xi is a tensor of size `len(obs) x num_states x num_states`. It contains
        the marginal probability of transitioning from i to j at t.
        """
        alpha = self.forward(obs)
        logprob = scipy.special.logsumexp(alpha[-1]+clip)
        beta = self.backward(obs)
        gamma = np.exp(alpha + beta - logprob)
        xi = np.zeros((len(obs)-1, self.num_states, self.num_states))
        for t in range(len(obs)-1):
          construction_matrix = np.matrix(alpha[t, :]).T + beta[t+1, :]
          partial = np.multiply(np.exp(construction_matrix - logprob), self.A)
          xi[t] = np.multiply(partial, self.B[:, obs[t+1]])
```

```
        return logprob, xi, gamma

        """
        SANITY CHECK

        The most straightforward way of implementing the forward, backward, and
        forward_backward methods would be to iterate through all the values and
        use the formulas in the slides to calculate the corresponding values.

        However, this may not be fast enough. If your model is taking too long
        to train, consider how you may speed up your code by reducing the number
        of for loops involved. How can you reformulate your code using matrix
        operations?

        Hint: we were able to implement each of the forward, backward, and
        forward_backward operations using only one for loop.
        """

    def learn_unsupervised(self, corpus, num_iters, print_every=10):
        """Run the Baum Welch EM algorithm

        corpus: the data to learn from
        num_iters: the number of iterations to run the algorithm
        print_every: how often to print the log-likelihood while the model is
        updating its parameters.
        """

        for i_iter in tqdm.trange(num_iters):
            """
            expected_si: a vector of size (num_states,) where the i-th entry is
            the expected number of times a sentence is transitioning from state
            i to some other state.

            expected_sij: an array of size (num_states, num_states) where the
            (i,j)-th entry represents the expected number of state transitions
            between state i and state j.

            expected_sjwk: an array of size (num_states, num_words) where the
            (j,k)-th entry represents the expected number of times the word w_k
            appears when at state j.

            expected_q1: a vector of size (num_states,) where the i-th entry is
            the expected number of times state i is the first state.

            total_logprob: The log of the probability of the corpus being
            generated with the current parameters of the HMM.
            """
            expected_si = np.zeros(self.num_states)
            expected_sij = np.zeros((self.num_states, self.num_states))
            expected_sjwk = np.zeros((self.num_states, self.num_words))
            expected_q1 = np.zeros(self.num_states)
            total_gamma = np.zeros(self.num_states)
```

```
        total_logprob = 0

        for review in corpus:
            logprob, xi, gamma = self.forward_backward(review)
            e_si = np.sum(gamma[:-1, :], axis=0)   # ok
            e_sjwk = np.zeros((self.num_states, self.num_words))
            expected_si = expected_si + e_si
            e_sij = np.sum(xi, axis=0) #
            expected_sij = expected_sij + e_sij
            for word in self.symbols:
                e_sjwk[:, word] = np.sum(gamma[np.asarray(review) == word], axis=0)
            expected_sjwk = expected_sjwk + e_sjwk
            e_q1 = gamma[0, :]
            expected_q1 = expected_q1 + e_q1
            total_logprob += logprob
            total_gamma = total_gamma + np.sum(gamma, axis=0)
        if i_iter % print_every == 0:
          print("log-likelihood", total_logprob)

        """
        The following variables should be the new values of self.A, self.B,
        and self.pi after the values are updated.
        """
        #A_new = (expected_sij.T/expected_si).T
        A_new = expected_sij / expected_si.reshape((self.num_states, 1))
        B_new = (expected_sjwk.T/total_gamma).T
        pi_new = expected_q1
        self.A = A_new
        self.B = B_new
        self.pi = pi_new/np.sum(pi_new)
```

Test Cases

The following are test cases that are meant to help you debug your code. The code involves six test suites - an initialization test, a forward test, a backward test, a forward_backward test, a baum_welch_update test, and a final end_to_end test.

```
def init_test():

    num_states = np.random.randint(100)
    num_words = np.random.randint(100)
    model = HMM(num_states, num_words)

    assert model.A.shape == (num_states, num_states)
    assert model.B.shape == (num_states, num_words)
    assert model.pi.shape == (num_states, )

    assert np.linalg.norm(np.sum(model.A, axis=1) - np.ones(num_states)) < 1e-10
```

```
    assert np.linalg.norm(np.sum(model.B, axis=1) - np.ones(num_states)) < 1e-10
    assert np.linalg.norm(np.sum(model.pi) - 1) < 1e-10

def forward_test():
    model = HMM(2, 10)
    model.A = np.array([[0.79034887, 0.20965113],
                        [0.66824331, 0.33175669]])
    model.B = np.array([[0.08511814, 0.06627238, 0.08487461, 0.15607959, 0.00124582, 0.129846
                        [0.18425462, 0.14326559, 0.14026994, 0.0215989,  0.17687124, 0.046812
    model.pi = np.array([0.77480039, 0.22519961])
    obs = [1, 8, 0, 0, 3, 4, 5, 2, 6, 3, 7, 9]
    alpha = model.forward(obs)

    print("The result of the forward function should be", np.array([[-2.96913, -3.43382],
                                                                     [ -4.66005, -9.19418],
                                                                     [ -7.35001, -7.89695],
                                                                     [ -9.65069, -9.95363],
                                                                     [-11.25815, -14.27392],
                                                                     [-18.14079, -14.4781 ],
                                                                     [-16.89275, -18.62696],
                                                                     [-19.45549, -20.17289],
                                                                     [-21.53772, -23.283  ],
                                                                     [-23.4927, -26.69119],
                                                                     [-25.84891, -26.73817],
                                                                     [-28.12237, -29.92402]]))
    print("Your value of alpha is:", np.round(alpha, 5))

def backward_test():
    model = HMM(2, 10)
    model.A = np.array([[0.79034887, 0.20965113],
                        [0.66824331, 0.33175669]])
    model.B = np.array([[0.08511814, 0.06627238, 0.08487461, 0.15607959, 0.00124582, 0.129846
                        [0.18425462, 0.14326559, 0.14026994, 0.0215989,  0.17687124, 0.046812
    model.pi = np.array([0.77480039, 0.22519961])
    obs = [1, 8, 0, 0, 3, 4, 5, 2, 6, 3, 7, 9]
    beta = model.backward(obs)

    print("The result of the backward function should be", np.array([[-25.42937, -25.58918],
                                                                      [-23.32164, -23.19959],
                                                                      [-21.11007, -21.02033],
                                                                      [-18.82215, -18.94381],
                                                                      [-16.78523, -16.33951],
                                                                      [-13.42847, -13.51924],
                                                                      [-11.24815, -11.19161],
                                                                      [ -8.88679,  -8.96441],
                                                                      [ -6.57374,  -6.70985],
                                                                      [ -4.51873,  -4.47419],
                                                                      [ -2.44529,  -2.51463],
                                                                      [  0, 0]]))

    print("Your value of beta is:", np.round(beta, 5))
```

```
def forward_backward_test():
    model = HMM(2, 10)
    model.A = np.array([[0.79034887, 0.20965113],
                        [0.66824331, 0.33175669]])
    model.B = np.array([[0.08511814, 0.06627238, 0.08487461, 0.15607959, 0.00124582, 0.129846
                        [0.18425462, 0.14326559, 0.14026994, 0.0215989,  0.17687124, 0.046812
    model.pi = np.array([0.77480039, 0.22519961])
    obs = [1, 8, 0, 0, 3, 4, 5, 2, 6, 3, 7, 9]
    logprob, xi, gamma = model.forward_backward(obs)

    print("The value of logprob should be:", -27.9693)
    print("Your value of logprob is:", np.round(logprob, 5))

    print("The value of xi should be:", np.array([[[0.64523, 0.00601],
                                                  [0.34278, 0.00598]],

                                                 [[0.60684, 0.38117],
                                                  [0.00551, 0.00648]],

                                                 [[0.40595, 0.2064 ],
                                                  [0.19863, 0.18902]],

                                                 [[0.5718,  0.03278],
                                                  [0.35711, 0.03831]],

                                                 [[0.02625, 0.90266],
                                                  [0.00109, 0.07   ]],

                                                 [[0.02482, 0.00251],
                                                  [0.81777, 0.15489]],

                                                 [[0.59943, 0.24316],
                                                  [0.08947, 0.06793]],

                                                 [[0.6143,  0.07461],
                                                  [0.25347, 0.05762]],

                                                 [[0.8357,  0.03207],
                                                  [0.12337, 0.00886]],

                                                 [[0.69872, 0.26034],
                                                  [0.02412, 0.01682]],

                                                 [[0.63701, 0.08583],
                                                  [0.22134, 0.05582]]]))
    print("Your value of xi is:", np.round(xi, 5))

    print("The value of gamma should be:", np.array([[0.65124, 0.34876],
                                                     [0.98802, 0.01198],
                                                     [0.61235, 0.38765],
                                                     [0.60458, 0.39542]
```

```
                                                    [0.60458, 0.39542],
                                                    [0.92891, 0.07109],
                                                    [0.02733, 0.97267],
                                                    [0.8426,  0.1574 ],
                                                    [0.68891, 0.31109],
                                                    [0.86777, 0.13223],
                                                    [0.95906, 0.04094],
                                                    [0.72284, 0.27716],
                                                    [0.85835, 0.14165]]))

        print("Your value of gamma is:", np.round(gamma, 5))

    def baum_welch_update_test():
        model = HMM(4, 10)

        model.A = np.array([[0.05263151, 0.62161178, 0.06683182, 0.25892489],
                            [0.26993274, 0.13114741, 0.32305468, 0.27586517],
                            [0.2951958,  0.14576492, 0.22474111, 0.33429817],
                            [0.29586018, 0.26065884, 0.1977772,  0.24570378]])

        model.B = np.array([[0.01800425, 0.09767131, 0.17824799, 0.12586453, 0.19514548, 0.054331
                            [0.04512782, 0.09469685, 0.1426164,  0.13851362, 0.08717793, 0.171525
                            [0.11055806, 0.10592473, 0.0051817,  0.07721441, 0.21761783, 0.203231
                            [0.08711377, 0.16703645, 0.0706214,  0.05297571, 0.10486868, 0.167945

        model.pi = np.array([0.21186864, 0.27156561, 0.37188523, 0.14468051])

        corpus = np.array([[7,3,2,5,0,3,2,9,4,2], [7,3,2,4,2,8,7,5,0,8], [7,3,2,3,1,7,3,8,6,7], [

        model.learn_unsupervised(corpus, 200)

        print("hmm.A should be", np.array([[0, 1, 0, 0],
                                          [0.14122, 0, 0.27099, 0.58779],
                                          [0.20671, 0, 0, 0.79329],
                                          [0, 0.90909, 0.09091, 0]]))
        print("Your implementation has hmm.A to be", np.round(model.A, 5))

        print("hmm.B should be", np.array([[0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
                                           [0.0625, 0, 0, 0.5, 0, 0.125, 0.125, 0, 0.125,
                                           [0, 0.20671, 0, 0, 0.79329, 0, 0, 0, 0, 0],
                                           [0.24667, 0, 0.57555, 0, 0.09556, 0, 0, 0, 0.08
        print("Your implementation has hmm.B to be", np.round(model.B, 5))

        print("hmm.pi should be", np.array([1, 0, 0, 0]))

        print("Your implementation has hmm.pi to be", np.round(model.pi, 5))

    def end_to_end_test():
        # Test Case 1

        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], [1,2,1,2,1,2,1
        hmm = HMM(num_states=2,num_words=4)
```

```
    hmm.learn_unsupervised(corpus, 10)
    print("After this test case, hmm.A should either be approximately,",  str(np.array([[0, 1
    print("This is your current value of hmm.A: ", np.round(hmm.A, 5))

    print("After this test case, hmm.B should either be approximately,", str(np.array([[0, 0,
    print("This is your current value of hmm.B: ", np.round(hmm.B, 5))

    # Test Case 2

    corpus = np.array([[0,0,0,0,0,0,0,0,0,0], [1,1,1,1,1,1,1,1,1,1], [2,2,2,2,2,2,2,2,2,2]])
    hmm = HMM(num_states=3, num_words=3)
    hmm.learn_unsupervised(corpus, 100)
    print("After this test case, hmm.A should be the identity matrix", np.eye(3))
    print("This is your current value of hmm.A: ", np.round(hmm.A, 5))

    print("After this test case, hmm.B should be some 3 by 3 permutation matrix")
    print("This is your current value of hmm.B: ", np.round(hmm.B, 5))
```

Test

To actually run the test cases, run the cell below:

```
init_test()
forward_test()
backward_test()
forward_backward_test()
baum_welch_update_test()
end_to_end_test()

"""
Note: The end_to_end_test is not as robust due to it using random starts. Try
running the test case a few times to see if you get a good result at least a few
times before deciding that your code is buggy.
"""
```

Training

Train a model:

```
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
```

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

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

```
<unk> 0.0008400594048061218
it 0.0013157268669496509
a 0.0015264385527403608
? 0.004489019888269412
! 0.16853773006426226
. 0.7995949018628039

state 3
, 0.0015229846841632395
disappointed 0.0015597360270125443
gum 0.0017554183110007652
; 0.003084453633128888
again 0.004715765065216035
a 0.004883506988123105
? 0.009155447641622103
<unk> 0.027699078400695267
! 0.14778298183563388
. 0.7534417799549227

state 4
thanks 0.0005963965602393657
gum 0.0006175622225634658
flavor 0.0006224876397629271
2 0.0006472306391509104
disappointed 0.0011985264103042041
a 0.0030061888155538804
<unk> 0.009099439774258136
? 0.013840174552267723
! 0.19193346170569542
. 0.7579232612539799

state 5
flavor 0.0010668381302601276
, 0.001186882003723401
disappointed 0.0012220563734887874
too 0.0016809769283933828
<unk> 0.0026764512120017033
again 0.002902523113770964
it 0.003317864843122423
? 0.004616107260821868
! 0.1451000649314592
. 0.8032049226746842

state 6
```

```
very 0.007931255697440647
not 0.007940320765410808
, 0.010251037296097781
good 0.010463309770952495
product 0.012868757260613385
a 0.014323303806946058
? 0.014383245637595962
. 0.02861919638082288
<unk> 0.10979784859546125
! 0.49409849137103123

state 7
after 0.0006325539691989234
review 0.0006372838488444298
shipping 0.0006712001506374481
it 0.0006796528617154521
```

We can also look at some samples from the model!

```
hmm.pi = hmm.pi/np.sum(hmm.pi)
for i in range(10):
    print(tokenizer.de_tokenize([hmm.generate(10)]))
```

```
['website the good prefer <unk> high com <unk> <unk> best']
['than green . ! ! ! t ! like .']
["! ! ! after <unk> amazon's ! ! ! lol"]
['! ! ! ! quality <unk> ! like ! very']
['! . . ! . . . . ! .']
['ship much d . . . . . . full']
['shipping ! much ! ! ! ! ! . again']
['! much dogs ! ! free <unk> ! <unk> !']
['! ! no . ! care <unk> . ! ship']
['! <unk> ! . prefer ! <unk> ! . !']
```

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.)

```
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):
  pred_ys = model.predict(xs_featurized)
  print("test accuracy", np.mean(pred_ys == ys))

def training_experiment(name, featurizer, n_train):
    print(f"{name} features, {n_train} examples")
    train_xs = np.array([
```

```
        hmm_featurizer(review)
        for review in tokenizer.tokenize(train_reviews[:n_train])
    ])
    train_ys = train_labels[:n_train]
    test_xs = np.array([
        hmm_featurizer(review)
        for review in tokenizer.tokenize(test_reviews)
    ])
    test_ys = test_labels
    model = train_model(train_xs, train_ys)
    eval_model(model, test_xs, test_ys)
    print()

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

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

```
hmm features, 100 examples
test accuracy 0.544
```

Experiments for Part 3

Restraining the sequences that can be created: vocabulary of size 4

In order to construct an HMM in order to restrict the generated sequences to these sequences, I decided to create 8 latent spaces: 01, 02, 12, 13, 20, 21, 30, 31. Each one of these $8 q_i$ will allow only the transition from q_{ij} to q_{ji} (encoded in the matrix transition A, which is a permutation matrix).

For the emission probabilities, the matrix B has been designed such as, if the hidden state is q_{ij}, I can only emit the token j.

Therefore, the HMM follows a dynamic: $q_{ij} \mapsto q_{ji}$ and emitting j, $q_{ji} \mapsto q_{ij}$ and emitting i and circling in this dynamics. It is therefore only determined by the initial distribution π, which assigns probabilities to states emitting 0 or 1, q_{i0} and q_{i1}.

```
model = HMM(8, 4)
```

```
model.A = np.array([[0, 0, 0, 0, 1, 0, 0, 0],
                    [0, 0, 0, 0, 0, 0, 1, 0],
                    [0, 0, 0, 0, 0, 1, 0, 0],
                    [0, 0, 0, 0, 0, 0, 0, 1],
                    [1, 0, 0, 0, 0, 0, 0, 0],
```

```
                    [0, 0, 1, 0, 0, 0, 0, 0],
                    [0, 1, 0, 0, 0, 0, 0, 0],
                    [0, 0, 0, 1, 0, 0, 0, 0]])

model.B = np.array([[0, 0, 1, 0], #02
                    [0, 0, 0, 1], #03
                    [0, 0, 1, 0], #12
                    [0, 0, 0, 1], #13
                    [1, 0, 0, 0], #20
                    [0, 1, 0, 0], #21
                    [1, 0, 0, 0], #30
                    [0, 1, 0, 0]])#31

model.pi = np.array([0, 0, 0, 0, 1/4, 1/4, 1/4, 1/4])
```

The correctness of this HMM is guarantedd by A and B, but for the sake of completeness:

```
target = [[0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2],
          [0, 3, 0, 3, 0, 3, 0, 3, 0, 3, 0, 3],
          [1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2],
          [1, 3, 1, 3, 1, 3, 1, 3, 1, 3, 1, 3]]

for i in range(10):
    print(model.generate(10))
```

```
[1, 3, 1, 3, 1, 3, 1, 3, 1, 3]
[0, 2, 0, 2, 0, 2, 0, 2, 0, 2]
[0, 2, 0, 2, 0, 2, 0, 2, 0, 2]
[0, 3, 0, 3, 0, 3, 0, 3, 0, 3]
[1, 3, 1, 3, 1, 3, 1, 3, 1, 3]
[0, 2, 0, 2, 0, 2, 0, 2, 0, 2]
[0, 2, 0, 2, 0, 2, 0, 2, 0, 2]
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
[1, 3, 1, 3, 1, 3, 1, 3, 1, 3]
[0, 3, 0, 3, 0, 3, 0, 3, 0, 3]
```

Information captured by Hidden States

In order to compare the information encoded in hidden states when varying the number of hidden states, I am going to train the entire HMM model on the corpus, with different number of states (2, 10 and 100) and try to qualitatively analyze the composition of the most probable words in these states. I chose to work on the entire corpus and not selecting a subset of the training portion because I wanted to capture important patterns in the data.

```
def visualize_words(model, review_tk, review, num_review):
  """This function analyses num_review reviews from the input reviews, and labels every word
```

```
    labellized_review = ""
    samples = np.random.randint(0, len(review_tk), num_review)
    selected_reviews = np.array(review_tk)[samples]
    reviews = np.array(review)[samples]
    for i, r in enumerate(selected_reviews):
      print(reviews[i])
      labellized_review = ""
      for token in r:
        word = tokenizer.token_to_word[token]
        hidden_state = np.argmax(model.B[:, token])  # this is an array of len(vocab_size, wher
        labellized_review += word + '(' + str(hidden_state) + ')' + ' '
      print(labellized_review)
```

With 2 states

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

```
 10%|█         | 1/10 [02:37<23:38, 157.56s/it]log-likelihood -366088.4838022064
100%|██████████| 10/10 [25:55<00:00, 155.58s/it]
```

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

```
state 0
not 0.0024101695634024057
the 0.002417013546234707
good 0.002424612492272223
product 0.003018897059185312
, 0.0034673125145925445
a 0.006306449340951357
? 0.015658984660505075
<unk> 0.04663799762747081
! 0.16826543348723572
. 0.6379116615071249

state 1
buy 0.0007041058962444358
it 0.0007309417439893712
review 0.0007658252708762268
too 0.0008922584113515918
a 0.0022494421980371704
<unk> 0.002601559022701181
again 0.0028219883252322935
? 0.005803555644962852
! 0.16283513684544748
. 0.7917045300577857
```

```
visualize_words(hmm2, train_reviews_tk, train_reviews, 2)
```

```
I am hooked on Stevia. Not only is it all natural but it tastes great and is truly a gu
i(0) am(0) hooked(0) on(0) stevia(0) .(1) not(0) only(1) is(0) it(0) all(0) natural(0)
I didn't mind the taste of this very much - it tasted like Minute Maid OJ mixed with so
i(0) didn't(0) mind(0) the(0) taste(0) of(0) this(0) very(0) much(0) it(0) tasted(0) li
```

I didn't mind the taste of this very much - it tasted like Minute Maid OJ mixed with soda. My husband couldn't stand it at all. My only issue with this was that it was a heavy-syrup sort of flavor that was a little overwhelming. If there was a better balance of flavor to liquid to make it a little lighter tasting, I think this would be much more refreshing and enjoyable. Might be fine for kids though - since they like that sugary orange stuff.

i(0) didn't(0) mind(0) the(0) taste(0) of(0) this(0) very(0) much(0) it(0) tasted(0) like(0) minute(1) (0) (0) mixed(0) with(0) soda(1) .(1) my(0) husband(1) couldn't(1) stand(0) it(0) at(0) all(0) .(1) my(0) only(1) issue(1) with(0) this(0) was(0) that(0) it(0) was(0) a(0) heavy(0) syrup(1) sort(0) of(0) flavor(1) that(0) was(0) a(0) little(0) overwhelming(1) .(1) if(0) there(0) was(0) a(0) better(1) balance(1) of(0) flavor(1) to(0) liquid(1) to(0) make(0) it(0) a(0) little(0) lighter(0) tasting(1) ,(0) i(0) think(0) this(0) would(0) be(0) much(0) more(0) refreshing(1) and(0) enjoyable(1) .(1) might(1) be(0) fine(0) for(0) kids(0) though(0) since(1) they(0) like(0) that(0) sugary(0) orange(0) stuff(1) . (1)

According to the previous analysis, we can infer that the learned hidden states seem to encode the following pattern:

- Hidden state (0) seems to encode the **action** terms : where, who, when, what ? Indeed, we can from the previous example that most of action terms seem to be encoded within the hidden state (1). This is quite corroborated by the probabilities of emission when being in state (1)
- Hidden state (1) seems to contain the nouns and transition words.

So it seems that, with only two hidden states, the separation is quite **syntactic**: hidden state (1) is for noun-related words and hidden state (0) would be more for action-related words. Moreover, we can see that there is somewhat a **class imbalance**: hidden state (0) seems very much more represented than hidden state (1).

With 10 states

```
%%time
hmm10 = HMM(num_states=10, num_words=tokenizer.vocab_size)
hmm10.learn_unsupervised(train_reviews_tk, 10)
```

```
 10%|█         | 1/10 [02:35<23:16, 155.13s/it]log-likelihood -361351.8665010071
100%|██████████| 10/10 [25:42<00:00, 154.21s/it]CPU times: user 25min 48s, sys: 46.9 s,
Wall time: 25min 42s
```

```
for i in range(hmm10.num_states):
    most_probable = np.argsort(hmm10.B[i, :])[-10:]
    print(f"state {i}")
    for o in most_probable:
        print(tokenizer.token_to_word[o], hmm10.B[i, o])
    print()
```

```
again 0.0016451048746894284
a 0.0017306251548540987
? 0.002619751545221334
<unk> 0.0044174466930776615
! 0.12113329106476406
. 0.8513318427194965

state 2
lol 0.0007794480894093742
, 0.0008899636800082337
flavor 0.0010020094152215827
disappointed 0.0011829875503439754
a 0.0018253916750608369
again 0.0034588682622226224
<unk> 0.01611923549032491
? 0.022410996304357274
! 0.36239859859312135
. 0.5598887483588093

state 3
it 0.0009572344137491842
recommend 0.0010199404667237313
thanks 0.001267316295577318
too 0.0013823587779058172
a 0.0016521559761242303
review 0.001692493127731792
<unk> 0.01139835804264475
? 0.015782030334310505
! 0.37354965235261867
. 0.555513552990175

state 4
again 0.0016869895937678785
buy 0.001690522040225976
lol 0.0019491229932316312
, 0.0025454789712366843
review 0.002850250306035709
; 0.0040205333090218565
a 0.0065786822882465115
<unk> 0.04259031719592784
! 0.41607687346605104
. 0.46639596737347033

state 5
```

```
state 5
disappointed 0.0004944885335533084
gum 0.0006023110000868821
flavor 0.0006200797755461466
review 0.0007565883692928646
again 0.0021734068339192576
a 0.0023987087284067406
<unk> 0.005724632332818013
? 0.009742255388297158
! 0.10144344682202702
. 0.8596382085852651

state 6
! 0.006983022819491699
, 0.007679896480082109
very 0.007984544679902113
not 0.007991561372399078
```

```
visualize_words(hmm10, train_reviews_tk, train_reviews, 1)
```

```
The product stinks and that's mildly stated.  The caps do not snap on easily and if you
the(6) product(6) <unk>(6) and(6) that's(7) <unk>(6) stated(0) .(8) the(6) <unk>(6) do(
```

The product stinks and that's mildly stated. The caps do not snap on easily and if you apply too much pressure the cup part caves in and you end up with ground coffee everywhere. I threw the whole order into the trash and called it a day. I will be using professionally produced K-cups (i.e, Dunkin Donuts, Starbucks, even Folgers) from this point on.

the(6) product(6) (6) and(6) that's(7) (6) stated(0) .(8) the(6) (6) do(2) not(6) (6) on(4) easily(8) and(6) if(6) you(6) (6) too(7) much(6) pressure(7) the(6) cup(3) part(7) (6) in(7) and(6) you(6) end(6) up(2) with(6) ground(6) coffee(6) ev

It is quite more complicated to analyze the information encoded into the learned hidden states with 10 hidden states. Although one mught infer that the hidden states seem to understand which words are paired together: too and pressure (coming from the expression too much pressure) come together in the hidden state (7). One interesting thing stemming from this visualization with 10 hidden states is that different hidden states share the same 'most probable' words. Let us see which ones those are.

```
from collections import Counter
to_count = []
for i in range(hmm10.num_states):
    most_probable = np.argsort(hmm10.B[i, :])[-10:]
    for o in most_probable:
      word = tokenizer.token_to_word[o]
      to_count.append(word)
print((Counter(to_count)))
```

```
Counter({'a': 10, '<unk>': 10, '!': 10, '.': 10, '?': 9, 'again': 7, ',': 5, 'review':
```

Is it something that we want ? Do we want different hidden states to have high probability to sample the same word ?

In my intuition, we don't. This is due to the fact that ultimately we might want to resolve the problem $Q = argmax_Q p(O|Q)$, ie decoding the sequence of hidden states that has produced a specific observation. In these computations, this involve (inside the *Viterbi Search*) $\delta_t(i)$ which will select the most probable hidden state at time step t. therefore, if a word has high probabilities of being generated by different sequences, there will be a problem (at the paroxysma of this case) of identifiability.

Also, the way I understand the latent space for the HMM we would like to have different hidden states that could be interpreted as latent clusters, and we don't want different clusters to share the same information (otherwise, we would not this this quantity of hidden states).

Therefore, I think that this information seems to say that having 10 hidden states is too much.

With 100 states

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

```
  0%|          | 0/10 [00:00<?, ?it/s]

 10%|█         | 1/10 [08:06<1:13:00, 486.72s/it]log-likelihood -354276.99156983424

 20%|██        | 2/10 [16:16<1:05:00, 487.51s/it]

 30%|███       | 3/10 [24:12<56:29, 484.24s/it]

 40%|████      | 4/10 [32:18<48:27, 484.61s/it]

 50%|█████     | 5/10 [40:21<40:20, 484.17s/it]

 60%|██████    | 6/10 [48:28<32:20, 485.01s/it]
```

```
 70%|███████   | 7/10 [56:36<24:18, 486.09s/it]

 80%|████████  | 8/10 [1:04:49<16:16, 488.05s/it]

 90%|█████████ | 9/10 [1:12:50<08:05, 485.98s/it]

100%|██████████| 10/10 [1:20:57<00:00, 485.79s/it]CPU times: user 1h 20min 43s, sys: 19
Wall time: 1h 20min 57s
```

```
for i in range(hmm100.num_states):
    most_probable = np.argsort(hmm100.B[i, :])[-10:]
    print(f"state {i}")
    for o in most_probable:
        print(tokenizer.token_to_word[o], hmm100.B[i, o])
    print()
```

```
state 0
d 0.0012741283953193283
; 0.0015860242243556368
it 0.001615200806281129
, 0.0020718921502826354
again 0.002309893516653986
? 0.008327101675735802
a 0.008389402948841388
<unk> 0.03640733816369366
! 0.0768090096133863
. 0.8238544510367533

state 1
a 0.0007101414586153364
d 0.0007468547972036924
too 0.0008646720693390415
review 0.0008677641510377434
! 0.0011656702171239076
it 0.001338394236078976
again 0.002179017371760319
? 0.01171556002481827
<unk> 0.019274054318202855
. 0.9393596135568875

state 2
shipping 0.0017713868984547982
lol 0.002297955317041309
disappointed 0.002848806527498697
it 0.0030023072115586457
again 0.007267426195775723
a 0.011664248216737094
<unk> 0.02462211085140608
? 0.027467102378026416
! 0.19077393927620342
```

```
. 0.6690598544091795

state 3
, 0.001940481897707122
gum 0.0020410684997683396
it 0.002196241335027324
product 0.002345787022229413
; 0.0033579179198647865
a 0.011632336122338426
? 0.03234778031211655
<unk> 0.03259729357778231
! 0.28434945500444986
. 0.5677536579176483

state 4
very 0.01768546155239918
the 0.01769933282495237
nice 0.020973428924488568
br 0.02672743190254157
did 0.0270243144208597
product 0.03161948346778137
much 0.03308777960248841
not 0.04493073930422865
<unk> 0.1250671652871821
. 0.14979160649184314
```

```
visualize_words(hmm100, train_reviews_tk, train_reviews, 1)
```

```
s and Rice Tortilla Chips are really tasty. I shared a few bags with my daughter in Iraq
74) beans(90) and(38) rice(74) tortilla(32) chips(38) are(88) really(4) tasty(81) .(20)
```

```
from collections import Counter
to_count = []
for i in range(hmm100.num_states):
    most_probable = np.argsort(hmm100.B[i, :])[-10:]
    for o in most_probable:
      word = tokenizer.token_to_word[o]
      to_count.append(word)
print((Counter(to_count)))
```

```
Counter({'.': 100, '<unk>': 99, '!': 97, '?': 96, 'a': 89, 'again': 71, ',': 61, 'it':
```

The same conclusions hold here.

Relationship between number of labeled examples and HMM representations

Here, we are going to use the same HMM with 10 hidden states for HMM representations.

First, what is a HMM representation ?

This reminds me of what is being done in the **LDA** paper, where we represented every distribution with a list of topics and their proportion in the document. Here, we summarize every review with the hidden_states estimated proportion inside the entire review.

For our representation, we will be using a HMM that has been trained with 10 hidden states, on the entire training corpus, for 10 updates of the Baum-Welch algorithm. Later on, we will see how the number of hidden states affect the performances of HMM-based representations.

Then, what is the classification task in this problem ?

It is a binary classification task, where we want to infer the positive/negative sentiment of a review based on the HMM representation of the review.

In a way, using HMM states distributions as sentence representations does make sense. Let's say for instance that our hidden states encode for positive/negative review (q_1 is positive and q_2 is negative). Then, our linear classifier will be able to learn that, since in the training phase, higher proportions of q_2 will be associated to a 0 label. Therefore, we see the importance of carefully selecting the number of hidden states.

However, there might be some issues over the fact that **we do not control what is encoded inside hidden states**. Indeed, this point is criically different from LDA, where we allocate our own topics and then the different documents draw topics from a global list of topics: here, the hidden states might be anything, they could be what is encoded in our experiment with 2 hidden states: syntactic structure, which will not help us towards our classification.

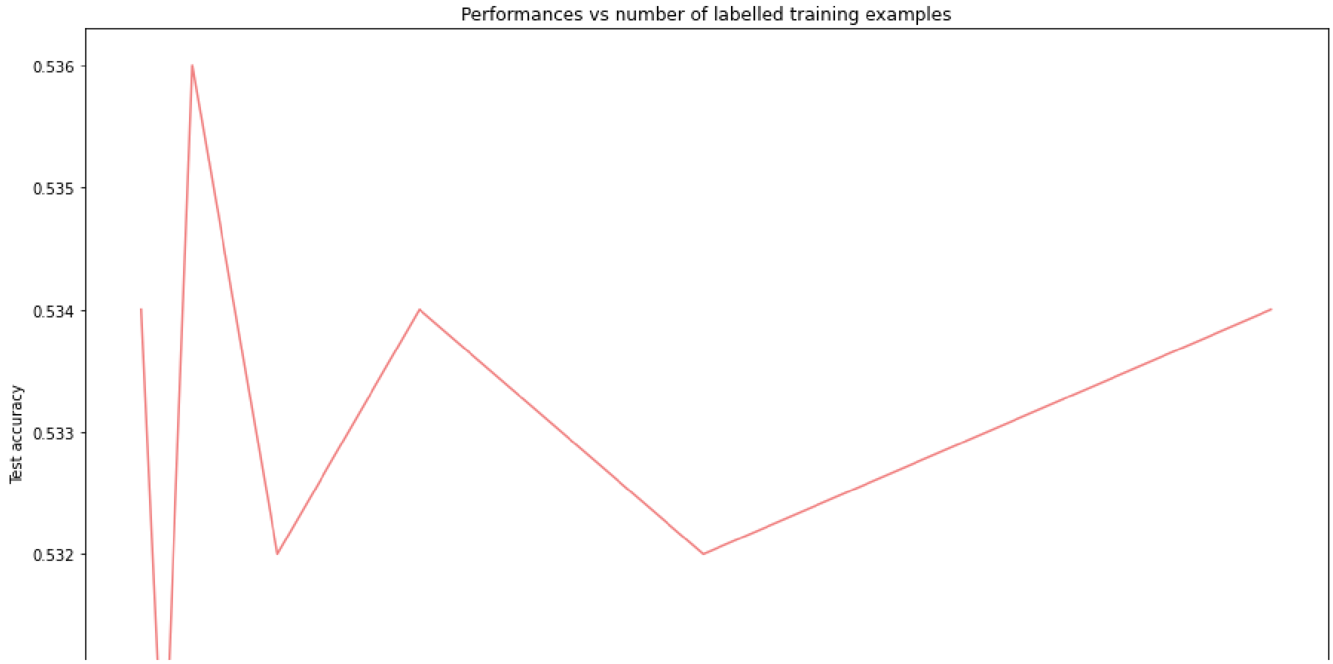
```
def train_model(xs_featurized, ys):
  import sklearn.linear_model
  model = sklearn.linear_model.LogisticRegression()
  model.fit(xs_featurized, ys)
  return model

def eval_model(model, xs_featurized, ys):
  pred_ys = model.predict(xs_featurized)
  return np.mean(pred_ys == ys)

def training_experiment(name, featurizer, n_train):
    train_xs = np.array([
        hmm_featurizer(review)
        for review in tokenizer.tokenize(train_reviews[:n_train])
    ])
    train_ys = train_labels[:n_train]
    test_xs = np.array([
        hmm_featurizer(review)
        for review in tokenizer.tokenize(test_reviews)
    ])
    test_ys = test_labels
```

```
    model = train_model(train_xs, train_ys)
    test_accuracy = eval_model(model, test_xs, test_ys)
    return test_accuracy

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

n_train = [3000]
for n in n_train:
  test = []
  for _ in range(10):
    test_accuracy = training_experiment("hmm", hmm_featurizer, n_train=n)
    test.append(test_accuracy)
  scores.append(test)
```

```
scores[:-1]
```

```
[[0.534, 0.534, 0.534, 0.534, 0.534, 0.534, 0.534, 0.534, 0.534, 0.534],
 [0.53, 0.53, 0.53, 0.53, 0.53, 0.53, 0.53, 0.53, 0.53, 0.53],
 [0.536, 0.536, 0.536, 0.536, 0.536, 0.536, 0.536, 0.536, 0.536, 0.536],
 [0.532, 0.532, 0.532, 0.532, 0.532, 0.532, 0.532, 0.532, 0.532, 0.532],
 [0.534, 0.534, 0.534, 0.534, 0.534, 0.534, 0.534, 0.534, 0.534, 0.534],
 [0.532, 0.532, 0.532, 0.532, 0.532, 0.532, 0.532, 0.532, 0.532, 0.532],
 [0.534, 0.534, 0.534, 0.534, 0.534, 0.534, 0.534, 0.534, 0.534, 0.534]]
```

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, figsize=(15, 10))
ax.plot([10, 50, 100, 250, 500, 1000, 2000], np.mean(np.array(scores[:-1]), axis=1), color='
plt.xticks([10, 50, 100, 250, 500, 1000, 2000])
ax.set_title('Performances vs number of labelled training examples')
ax.set_xlabel('Number of labelled training examples')
ax.set_ylabel('Test accuracy')
plt.show(fig)
```



Although not very substantially, we can see that icnreasing the number of labelled examples help classification thanks to HMM-based sentence representations.

Equivalence HMM bigrams

- Hypothesis fir Bigram Model: the likelihood of an observation $O_{1:T}$ is $p(O_{1:T}) = \prod_{t=1}^{T-1} p(o_{t+1}|o_t)$

Let's consider a HMM with v-states: every state q_i is corresponding to one word w_i. This means that $b_i(w_i) = 1$. Let $O_{1:T}$ be an observation. Under the HMM model, the likelihood of this observation is $p(O_{1:T}) = \sum_Q p(O|Q)p(Q)$. And we have

$$p(O_{1:T}/Q_{1:T}) = \prod_t p(o_t|q_t)$$

(HMM Hypothesis). Therefore, the product is 0 **except** for the very specific sequence where every q_i is corresponding to w_i (being the state for which $b_i(w_i) = 1$, for which the product is equal to 1. We have therefore $p(O_{1:T}) = p(Q^*) = \prod_{t=1}^{T-1} p(q_{t+1}^*|q_t^*)$, and since the state q_t is reduced to the observation o_t, both models are equivalent.