hmm_gmm

March 20, 2025

1 Python demo of Forced Align with HMM-GMM

This demo walks through how to use a Hidden Markov Model-Gaussian Mixture Model (HMM-GMM) architecture to perform Forced Align. I assume decent familiarity with Python and the Numpy library, though I try to explain the libraries and methods I use in my code as much as possible. I assume little machine learning background and explain models as best I can when they come up. However, this is not meant to be a tutorial in machine learning per se, but rather an illustration of how a particular family of models can be implemented in Python, with the goal of giving the reader the chance to play around with the inner workings of algorithms they may have only seen before displayed as neat, abstracted equations in ML textbooks that bely the complexity of their implementation and usage.

This notebook can be seen as a companion to chapters 6 and 9 of Juarafsky and Martin 2009, though I go about it in a somewhat different order, introducing Gaussian Mixture Models for modeling the acoustics of phones first and then introduce sequence modeling with Hidden Markov Models. I try to make my notation as close to theirs as possible, and also reference pages and sections from their textbook for further reading on the topics I discuss. By no means do I pretend to improve upon their explanation, I merely present a sandbox for demonstrating what they explain. I hope that the explanations I have left make this notebook self-contained, though I recommend referencing the Jurafsky and Martin book for deeper explanation as desired.

Code and resources for this demo can be found in this github repo. For purposes of the demo I have recorded and segmented myself producing five words consisting of the phones [, i, l, n], namely 'lawn', 'lean', 'kneel', 'knee', 'gnaw.' The audio is found in `ailn.wav`, segmentations in `ailn.TextGrid` and the data file in `ailn.csv`, with a Python script using `parselmouth` in `gather_measures.py` for generating the acoustic measures should you wish to record your own audio for use in the demo. The features used are the first three formants ('f1', 'f2', 'f3') and the amplitude envelope ('amp').

As a side note, to avoid the inconvenience of typing the IPA symbol [ ] repeatedly and because there is no [a] here in question to confuse it worth, I will write [a] instead of [ ] throughout this post.

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import torch
import seaborn as sb
torch.manual_seed(1337)
```

[1]:
```
<torch._C.Generator at 0x120d01cd0>
```

First, let's load in our dataset with Praat TextGrid data.

[2]:
```
csv_path = 'ailn.csv'
df = pd.read_csv(csv_path)
df
```

[2]:

	f1	f2	f3	amp	phone	word	word_ipa	\
0	308.842307	691.112245	2183.135206	66.923292	l	lawn	lan	
1	342.378196	734.724329	2186.841639	69.177411	l	lawn	lan	
2	362.149719	764.074480	2210.458974	70.539616	l	lawn	lan	
3	356.570667	762.745254	2225.582023	71.400740	l	lawn	lan	
4	357.828687	758.569292	2206.414437	72.209283	l	lawn	lan	
..	…	…	…	…	…	…	…	
345	755.145121	805.773587	2260.783925	75.332562	a	gnaw	na	
346	619.998312	805.533887	2206.163646	74.449697	a	gnaw	na	
347	585.264409	806.373248	2106.831388	73.476442	a	gnaw	na	
348	465.633125	818.948885	2016.998327	72.475936	a	gnaw	na	
349	436.861831	819.351235	2057.549432	71.271442	a	gnaw	na	

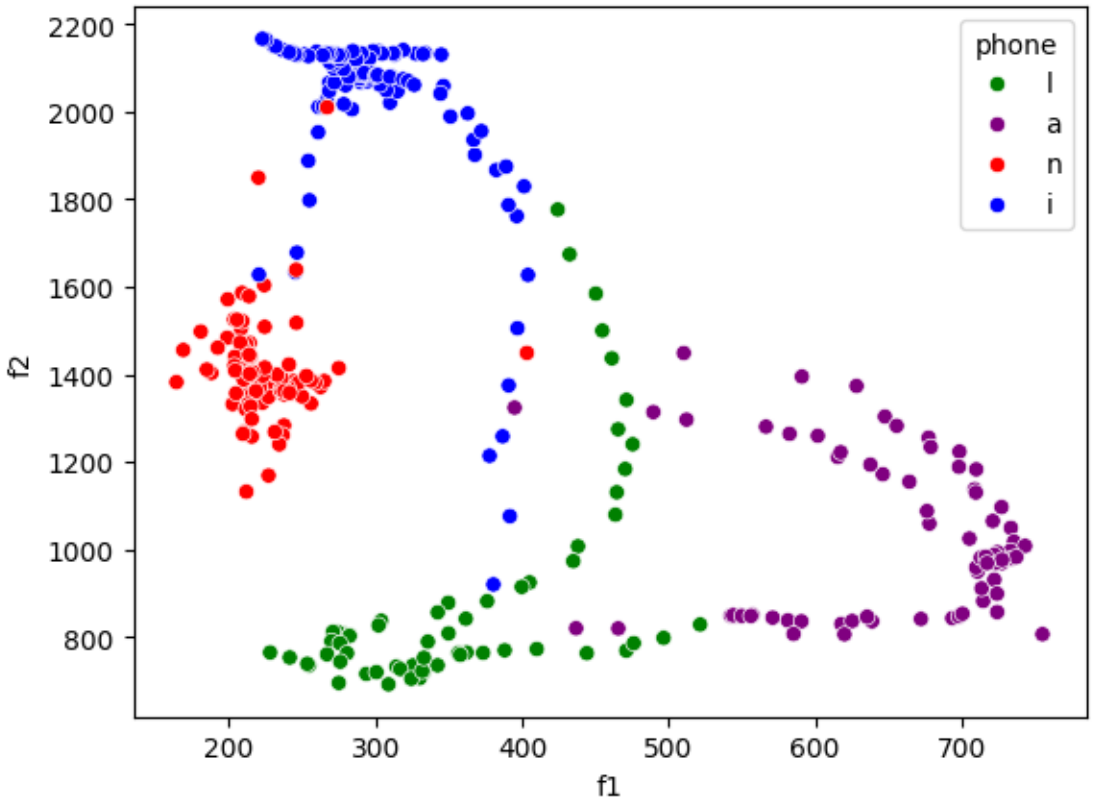
	time
0	1.939838
1	1.946088
2	1.952338
3	1.958588
4	1.964838
..	…
345	6.764838
346	6.771088
347	6.777338
348	6.783588
349	6.789838

[350 rows x 8 columns]

Let's plot our data. Note there are four features, so we'll only be able to view two at a time.

[3]:
```
# to make plotting easier
color_map = {
    'a': 'purple',
    'i': 'blue',
    'l': 'green',
    'n': 'red',
}
df['color']=df['phone'].map(color_map)
```

[4]:
```
# try swapping `x` and `y` for other features!
sb.scatterplot(data=df, x='f1', y='f2', hue='phone', palette=color_map)
plt.show()
```



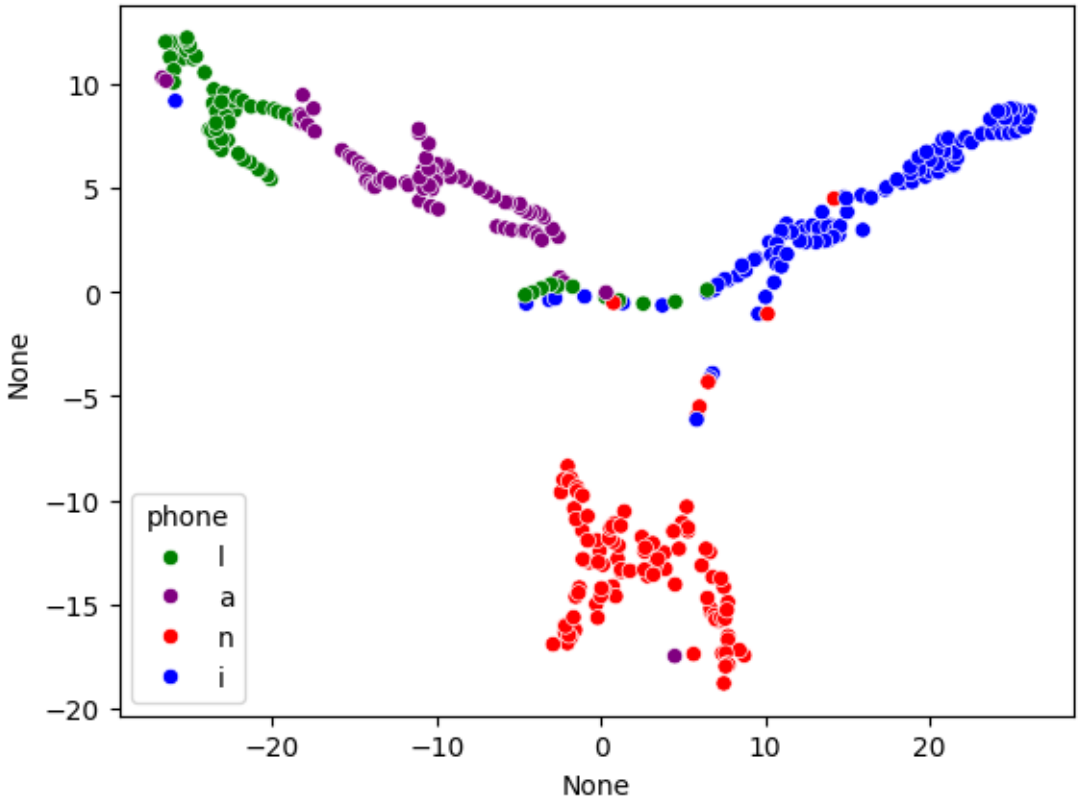
2 Dimensionality reduction

All of the methods we are using would work fine on our four-feature set. In fact, it's typical to use a much larger set of 39 features called MFCCs (Jurafsky & Martin 2009, Ch. 9, p 297). However, this is a simplified toy example, and so visualization is a high priority. For that reason, we'll reduce our four-feature set to two features using tSNE, a dimensionality reduction algorith. We can use the `TSNE` class from `sklearn.manifold` to easily reduce our data to two dimensions. While we're at it, let's split the data into features for each individual phone.

Note we convert the output of `TSNE` to a pytorch tensor since `pomegranate`, the library we will use for modeling HMMs, uses pytorch as a backend.

[5]:
```
from sklearn.manifold import TSNE

# define features, then shrink with TSNE
feat_cols = ['f1', 'f2', 'f3', 'amp']
X_4d = df[feat_cols].to_numpy()
```

```
X = TSNE().fit_transform(X_4d)
X = torch.tensor(X)

# store feature for each phone separately
a_mask = df['phone']=='a'
i_mask = df['phone']=='i'
l_mask = df['phone']=='l'
n_mask = df['phone']=='n'

a_feats = X[a_mask]
i_feats = X[i_mask]
l_feats = X[l_mask]
n_feats = X[n_mask]

sb.scatterplot(x=X[:,0], y=X[:,1], hue=df['phone'], palette=color_map)
plt.show()
```



3 Exploring separability

Let's get a grasp of how easy it is to classify the four phones in our dataset, but to do that we

We'll do this by performing k-means clustering, and seeing how well the resulting clusters match with our phones. Before we do that, let's save the phone labels to an array `Y` by converting each unique character to an integer from 0 to 3.

[6]:
```
# define labels
phones='ailn'
phone_labels = df['phone'].to_numpy()
Y=df['phone'].apply(phones.index).to_numpy()
np.array([*zip(phone_labels,Y)])
```

[6]:
```
array([['l', '2'],
       ['l', '2'],
       ['l', '2'],
       ['l', '2'],
       ['l', '2'],
       ['l', '2'],
       ['l', '2'],
       ['l', '2'],
       ['l', '2'],
       ['l', '2'],
       ['l', '2'],
       ['l', '2'],
       ['l', '2'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
```

```
['a', '0'],
['a', '0'],
['a', '0'],
['a', '0'],
['a', '0'],
['a', '0'],
['a', '0'],
['a', '0'],
['a', '0'],
['a', '0'],
['a', '0'],
['a', '0'],
['a', '0'],
['a', '0'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
```

```
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
```

```
['i', '1'],
['i', '1'],
['i', '1'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
```

```
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
```

```
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['l', '2'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
```

```
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['i', '1'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['n', '3'],
['a', '0'],
['a', '0'],
['a', '0'],
['a', '0'],
['a', '0'],
['a', '0'],
['a', '0'],
['a', '0'],
['a', '0'],
```

```
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0'],
       ['a', '0']], dtype='<U21')
```

Now let's try running k-means clustering on the data with random centroids. We can evaluate these centroids using `v_measure_score`, which measures the mutual information between the cluster identities and the labels. By using this metric, we can get an idea of how closely the clusters we generate map to phones regardless of whether the index assigned to each centroid matches the number representing the phone it is closest to.

[7]:
```
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score, v_measure_score

# fit model
kmeans = KMeans(n_clusters=4)
y_hat = kmeans.fit_predict(X)
v_measure_score(y_hat, Y)
```

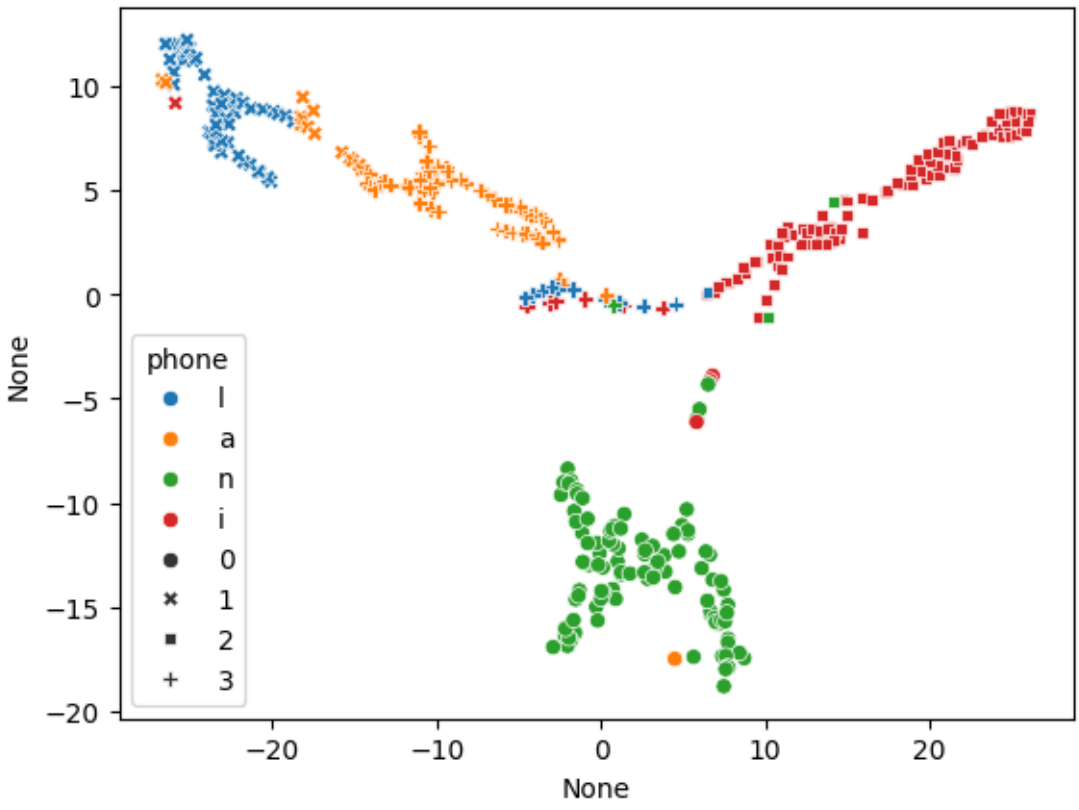
```
/Users/markjos/projects/forced_align_writeup/.venv/lib/python3.12/site-
packages/threadpoolctl.py:1226: RuntimeWarning:
```

```
Found Intel OpenMP ('libiomp') and LLVM OpenMP ('libomp') loaded at
the same time. Both libraries are known to be incompatible and this
can cause random crashes or deadlocks on Linux when loaded in the
same Python program.
Using threadpoolctl may cause crashes or deadlocks. For more
information and possible workarounds, please see
    https://github.com/joblib/threadpoolctl/blob/master/multiple_openmp.md

  warnings.warn(msg, RuntimeWarning)
```

[7]:
```
0.7217863187793011
```

Let's visualize the clusters and see how well they fit with phones.

[8]:
```
sb.scatterplot(
    x=X[:,0],
    y=X[:,1],
    hue=df['phone'],
    style=y_hat,
)
plt.show()
```



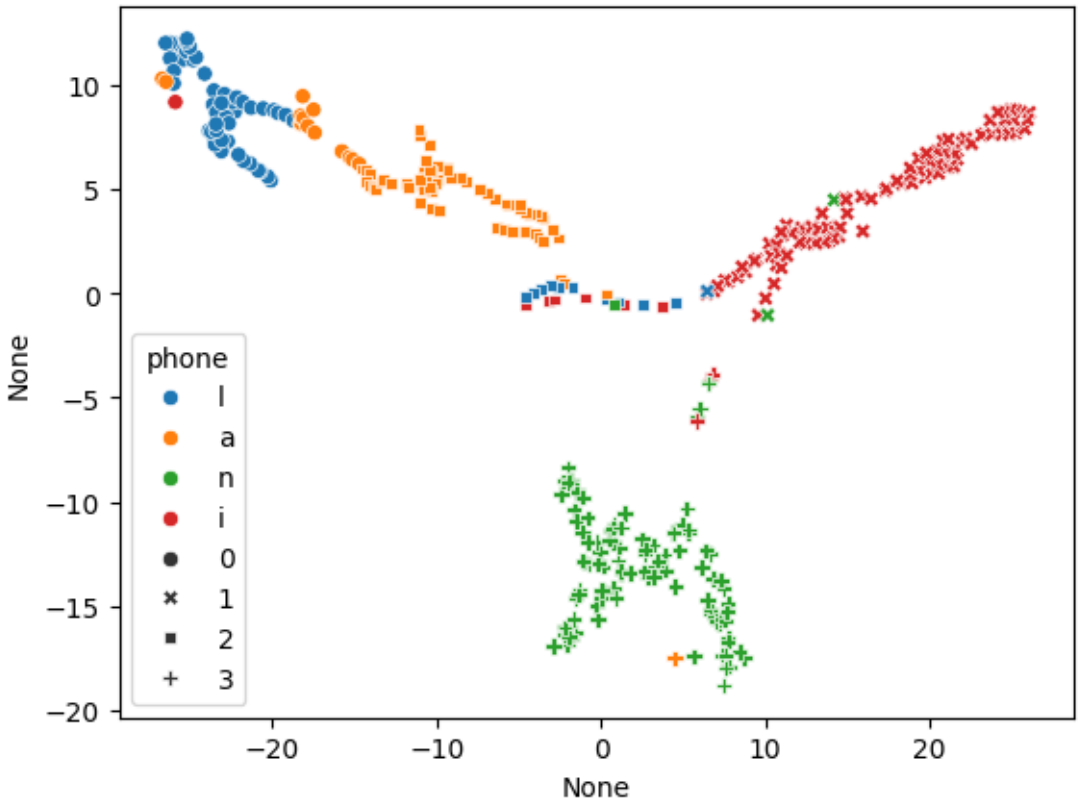
When I ran it, I got cluster 0 ≈ [n], cluser 1 ≈ [l], cluster 2 ≈ [i] and cluster 3 ≈ [a].

[9]:
```
# define centroids
sample_row = lambda m: m[torch.randint(0,len(m), (1,))]
sample_a = sample_row(a_feats)
sample_i = sample_row(i_feats)
sample_l = sample_row(l_feats)
sample_n = sample_row(n_feats)
start_centroids = torch.concat([sample_a, sample_i, sample_l, sample_n])

# fit model
kmeans_seeded = KMeans(init=start_centroids, n_clusters=4)
y_hat = kmeans_seeded.fit_predict(X)
accuracy_score(y_hat, Y)
```

[9]:
```
0.6371428571428571
```

Run it a few times. I get accuracies varying from 0.6-0.9. (**HINT/SPOILER** for the end of the notebook when we seed the Gaussians with k-means produced here, you'll want to have accuracy closer to 0.9, so run it until you get around there. It should just take 2 or 3 tries at most.)

[10]:
```
sb.scatterplot(
    x=X[:,0],
    y=X[:,1],
    hue=df['phone'],
    style=y_hat,
)
plt.show()
```



Other than having the right ordering of phones $[1, 2, 3, 4] \approx [a, i, l, n]$, the clusters look about the same as what we saw before.

4 Fitting phone models

Now let's try a different clustering method, namely Gaussian Mixture Models (GMMs). A Gaussian model is another name for a normal distribution. A GMM, then, is an ensemble or *mixture* of several normal distributions. By using multiple normal distributions, we can parameterize data which are not normally distributed as a whole. To see what this means, let's visualize fitting a single Gaussian versus two-Gaussian mixtures for each phone in our dataset.

This code uses the `pomegranate` library to fit a Gaussian Mixture Model for each phone. Later on, we'll see a bit more about what goes into fitting the parameters of a Gaussian model.

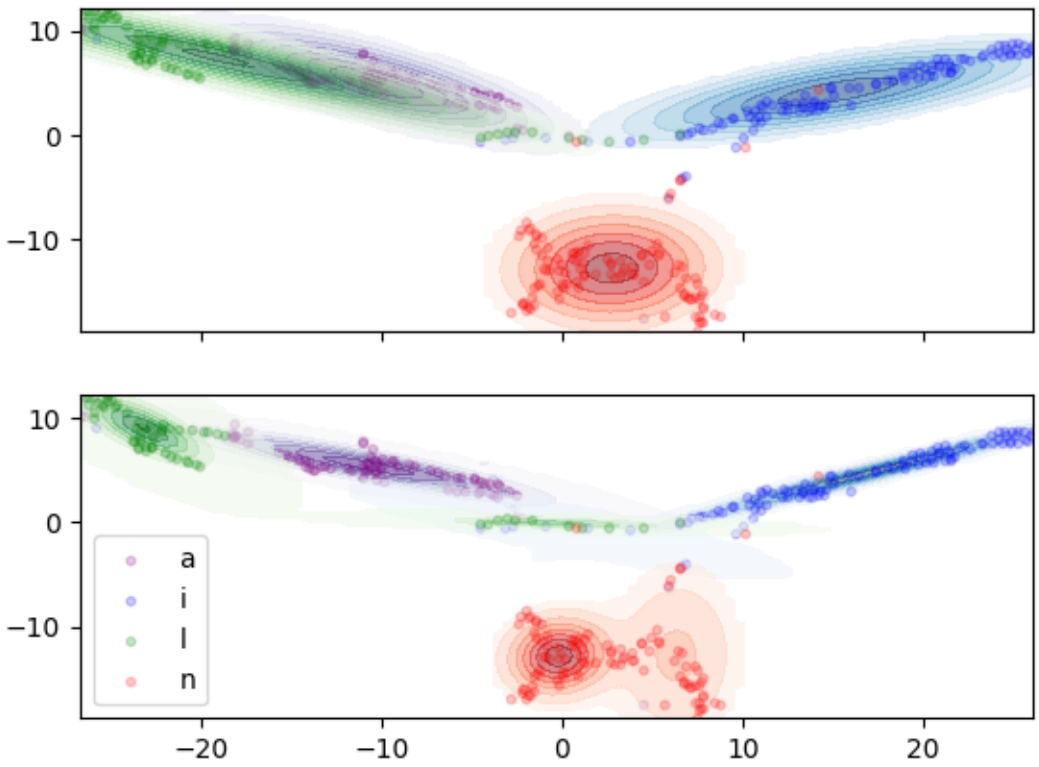
Warning: The `GeneralMixtureModel` call may return a `_LinAlgError` but I found that after running it a few times it resolves itself. The error likely depends on some random state.

[11]:
```
from pomegranate.gmm import GeneralMixtureModel
from pomegranate.distributions import *
import matplotlib.pyplot as plt
```

```
X_2d = X
phone_tuples = []
for phone, features, color in [
    ('a', a_feats, 'purple'),
    ('i', i_feats, 'blue'),
    ('l', l_feats, 'green'),
    ('n', n_feats, 'red')
]:
    bi_model = GeneralMixtureModel([Normal(), Normal()]).fit(features)
    uni_model = Normal().fit(features)
    phone_tuples.append((phone, features, color, bi_model, uni_model))
```

Now let's plot it with some lovely code stolen from the Pomegranate docs on mixture models.

[12]:
```
fig,axes=plt.subplots(2, sharex=True, sharey=True)

x_min = X_2d[:,0].min()
x_max = X_2d[:,0].max()
x = np.linspace(x_min, x_max, num=100)
y_min = X_2d[:,1].min()
y_max = X_2d[:,1].max()
y = np.linspace(y_min, y_max, num=100)

assert len(x)==100
assert len(y)==100

xx, yy = np.meshgrid(x, y)
x_ = np.array(list(zip(xx.flatten(), yy.flatten())))


for phone, features, color, bi_model, uni_model in phone_tuples:

    p1 = uni_model.probability(x_).reshape(len(x), len(y))
    p2 = bi_model.probability(x_).reshape(len(x), len(y))

    for prob, ax in zip([p1,p2], axes):
        # only show probability above 90th quantile, to minimize overlap
        quantile20 = prob.quantile(0.90)
        prob[prob<quantile20]=float('-inf')

        # ax.title("Single Gaussian", fontsize=12)
        ax.contourf(xx, yy, prob, cmap=color.capitalize()+'s', alpha=0.5)
        ax.scatter(features[:,0], features[:,1], s=10, color=color, alpha=0.2,␣
 ↪label=phone)
plt.legend()
plt.show()
```



Again, results will vary on your random state but you should see one Gaussian contour for each vowel in the top and twov in the bottom. It should be clear to see how the two-mixture model is able to represent the data better than the single Gaussian models, especially for [l] which has two discontuous clusters of points.

Now we've found a model which can accurately represent our phones, but this isn't enough to perform Forced Align. To do that, we need something which can model the acoustic sequence as a whole, rather than just considering a single time frame.

5 Hidden Markov Models

Hidden Markov Models (HMMs) are sequence models. To understand Hidden Markov Models, consider an example of a standard or 'observable' (non-hidden) Markov model (or Markov chain): a bigram language model. A bigram language model represents the probability of a word given the word immediately before it, $P(\text{word}|\text{previous\_word})$. If a language model yields $P(\text{game}|\text{fun}) = 0.2$, then given the word 'fun' in any text sequence, there is a 20% chance the next word is 'game.' Likewise, if $P(\text{homework}|\text{fun}) = 0.001$, then there is a 0.1% chance that the next word after 'fun' will be 'homework.'

Imagine we want to model a sequence of elements that we can't observe or measure directly, for example the phones that make up a spoken word or sentence. Given the audio waveform of speech, we cannot straightforwardly determine what series of phones the speaker was producing when making the utterance, else forced alignment and speech recognition would be trivial tasks. What we

can measure are the acoustic *observations*, e.g. the formant and intensity values used above. Since each acoustic observations was 'produced' by a given phone, and since different phones produce different manner of observations, we should be able to reconstruct the sequence of phones from a sequence of acoustic observations. These phones are the *hidden states* of the HMM, and when modeling an HMM we talk about hidden states *emitting* observations.

HMMs are characterized by two probabilities: transition probability between states i.e. $P(\text{state}|\text{previous\_state})$ and emission probabilities, i.e. $P(\text{observation}|\text{state})$. The former is directly analagous to the language model. We could say that a language model is a special HMM where the states (words) are observable, and thus we don't need any separate observations to reconstruct them. The latter, the emission probabilities, are represented by the Gaussian models we saw earlier. Each Gaussian model is a different state, [a], [i], [l] or [n]. The value that, e.g., the Gaussian model for [n] assigns to a given point on the plot is equivalent to $P(\text{point}|[\text{n}])$.

I've been using full words in my probability expressions so far, but for conciseness from now on I'll write O for the sequence of observations, o_t for the observation at time t, Q for the set of hidden states (following the notation in Jurafsky & Martin 2009), s_t for the hidden state at time t, and i and j will generally be used as indices pointing to individual states in the set of hidden states.

There are three crucial problems associated with HMMs (Jurafsky & Martin 2009, Ch 6 p. 179): 1. Likelihood assignment. What is the likelihood of some sequence of observations O given the HMM hidden states Q and parameters λ, i.e. what is $P(O|Q,\lambda)$? 2. Decoding. Given a sequence of observations O, what is the most likely sequence of hidden states Q that produced them, i.e. what Q maximizes $P(O|Q,\lambda)$? 3. Training. Given a sequence of observations O and hidden states Q, what parameters λ are optimal for describing the joint sequence, i.e. what λ maximizes $P(O|Q,\lambda)$?

Where 'parameters' refer to the transition and emission probabilities of the HMM.

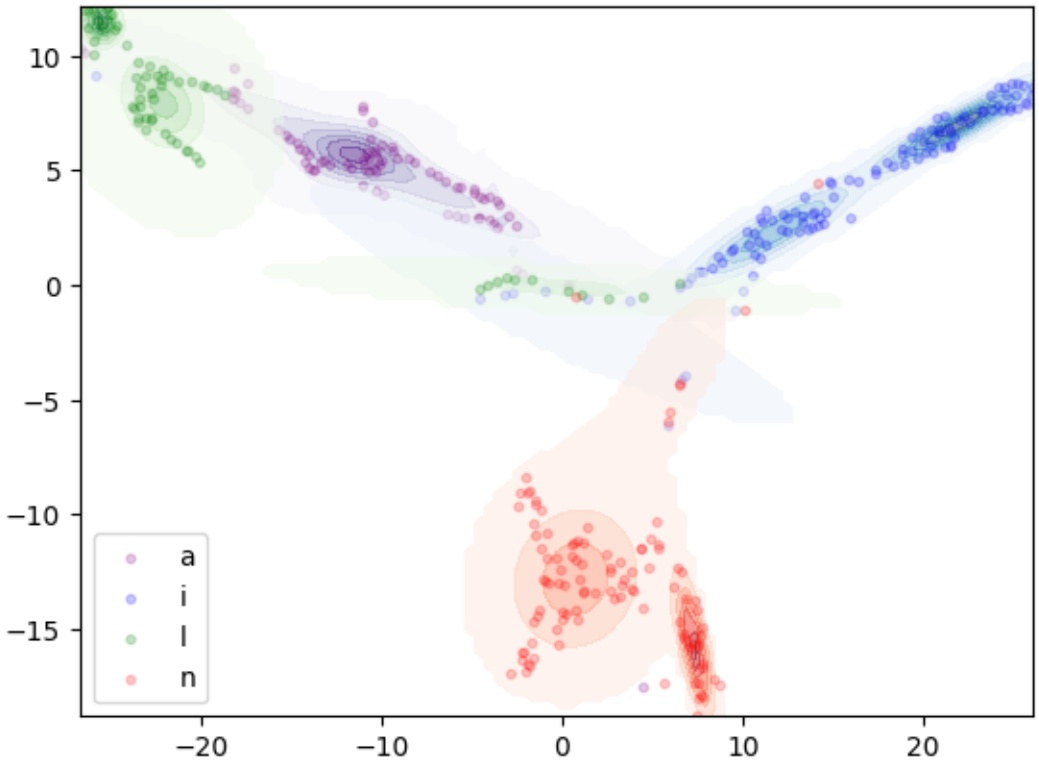
To begin, we'll load in and fit an HMM from the `pomegranate` to illustrate Forced Align inference. Later we'll construct our own `ToyHMM` class to peak into the math behind training HMMs.

First we instantiate a `DenseHMM` class. A `DenseHMM` is just an HMM with connections between all states. If we were making a very large HMM with only a small subset of state transitions allowed, using a `SparseHMM` would be more efficient computationally, but it's not necessary for our toy example. We add states to the HMM by passing GMM models to the `add_distributions()` function, such that each state has a GMM associated with it.

[13]:
```
from pomegranate.hmm import DenseHMM

hmm = DenseHMM()

phones = 'ailn'
states = []

for features in [a_feats, i_feats, l_feats, n_feats]:
    states.append(GeneralMixtureModel([Normal(), Normal(), Normal()]).
 ↪fit(features))
hmm.add_distributions(states)
```

Let's define a function so we can easily visualize the Gaussian distributions of a HMM-GMM.

[14]:
```
def plot_gaussians(states, X=X, Y=Y, phones='ailn', colors=['purple', 'blue',␣
 ↪'green', 'red']):
    x_min = X_2d[:,0].min()
    x_max = X_2d[:,0].max()
    x = np.linspace(x_min, x_max, num=100)
    y_min = X_2d[:,1].min()
    y_max = X_2d[:,1].max()
    y = np.linspace(y_min, y_max, num=100)

    assert len(x)==100
    assert len(y)==100

    xx, yy = np.meshgrid(x, y)
    x_ = np.array(list(zip(xx.flatten(), yy.flatten())))

    assert len(x)==100
    assert len(y)==100
    for state, phone, color in zip(states, phones, colors):
        prob = state.probability(x_).reshape(len(x), len(y))
        phone_X = X[Y==phones.index(phone)]

        # only show probability above 90th quantile, to minimize overlap
        quantile90 = prob.quantile(0.90)
        prob[prob<quantile90]=float('-inf')

        plt.contourf(xx, yy, prob, cmap=color.capitalize()+'s', alpha=0.5)
        plt.scatter(phone_X[:,0], phone_X[:,1], s=10, color=color, alpha=0.2,␣
 ↪label=phone)
    plt.legend()
    plt.show()
plot_gaussians(hmm.distributions)
```



Note that since we're working with hand-segmented data, we can define the distributions for each phone right off the bat, without needing any training. Later on we'll see how to train these distributions without needing pre-segmented data.

By adding the distributions we have defined our emission probabilities, but we haven't set any transition probabilities yet. We can set transition probabilities using the `add_edge()` function. For `pomegranate`, this function takes as arguments two distributions (representing the two states) and a float indicating the weight of the connection, equivalent to the transition probability between the two edges.

[15]:
```
for state1 in states:
    for state2 in states:
        hmm.add_edge(state1, state2, 0.25)
hmm.edges
```

[15]:
```
tensor([[-1.3863, -1.3863, -1.3863, -1.3863],
        [-1.3863, -1.3863, -1.3863, -1.3863],
        [-1.3863, -1.3863, -1.3863, -1.3863],
        [-1.3863, -1.3863, -1.3863, -1.3863]])
```

When we view the edges, we see the weights are not 0.25 but -1.3863. This is because `pomegranate` doesn't store the edge probability per se but the log of the edge probability.

We've just made a very naive assumption here by giving an equal chance of transitioning from

any one state to any other. For one, certain state sequences like [n,l] never occur in the corpus (e.g. there's no word like 'unload'). Furthermore, given how long phone intervals are relative to an individual audio frame, it's generally much more likely that a state will transition into itself than into any other state. Look at the phone sequence for 'lawn', for instance.

[16]:
```
lawn_sequence = df.loc[df['word']=='lawn', 'phone']
lawn_str = ''.join(lawn_sequence)
lawn_str
```

[16]:
```
'lllllllllllllaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaannnnnnnnnnnnnnnnnnnnnnnn'
```

Again, since we already have time alignments, calculating the transition probabilities is simply a matter of counting how many times a given state i transitions to another state j divided by all transitions out of i, as defined below (equation adapted from Jurafsky & Martin 2009, Ch. 6, p. 188):

$$P_{\text{transition}}(i|j) = \frac{\text{count}(i \to j)}{\sum_{s \in S} \text{count}(i \to s)}$$

As Jurafsky & Martin (2009) show, we can represent this as a matrix of transition probabilities a, where a_{ij} is equivalent to $P_{\text{transition}}(i|j)$. For sake of readability I will name the matrix a `transition_mat` in my code.

To calculate the `transition_mat`, we can start by counting all the transitions between states across the corpus and save them to an intermediary matrix `transition_counts`. To make this easier, let's add a column `'last_phone'` to our dataframe indicating the previous state before any row. We'll treat each word as a single state sequence.

Note that we use special states for the beginning and end of the sequence, indicated by the Regex characters `'^'` and `'$'`. We don't have rows for last states in the dataframes, so to be able to count transitions into the last states, we store a separate dataframe–we don't merge the two since last states have no observations, and later we'll want to be able to easily slice out all of the observations for a given word from the dataframe, whereas we won't need the end state rows again after counting transitions.

[17]:
```
words = df['word'].unique() # ['lawn', 'lean', 'kneel', 'knee', 'gnaw']
df['last_phone']=''
last_state_rows = []
for word in words:
    word_mask = df['word']==word
    phone_seq = df.loc[word_mask, 'phone']
    shifted_seq = np.insert(phone_seq, 0, '^')[:-1] # prepend '^' and cut off␣
 ↪last state
    df.loc[word_mask, 'last_phone']=shifted_seq
    last_state_row = {'phone': '$', 'last_phone': phone_seq.iloc[-1], 'word':␣
 ↪word}
    last_state_rows.append(last_state_row)
df.head()
```

[17]:

	f1	f2	f3	amp	phone	word	word_ipa	\
0	308.842307	691.112245	2183.135206	66.923292	l	lawn	lan	

```
1  342.378196  734.724329  2186.841639  69.177411     l  lawn      lan
2  362.149719  764.074480  2210.458974  70.539616     l  lawn      lan
3  356.570667  762.745254  2225.582023  71.400740     l  lawn      lan
4  357.828687  758.569292  2206.414437  72.209283     l  lawn      lan

       time  color last_phone
0  1.939838  green          ^
1  1.946088  green          l
2  1.952338  green          l
3  1.958588  green          l
4  1.964838  green          l
```

[18]:
```
last_state_df = pd.DataFrame(last_state_rows)
last_state_df.head()
```

[18]:
```
  phone last_phone   word
0     $          n   lawn
1     $          n   lean
2     $          l  kneel
3     $          i   knee
4     $          a   gnaw
```

Now we can iterate through each unique state and add their counts to the matrix. The matrix is of shape (6×6), i.e. 4 phones and two boundary states.

[19]:
```
transition_counts = torch.zeros((6,6))
state_names = '^ailn'
# count all non-final states
for i, state1 in enumerate(state_names):
    state1_mask = df['last_phone']==state1
    for j, state2 in enumerate(state_names):
        state2_mask = df['phone']==state2
        transition_counts[i,j]=len(df[state1_mask&state2_mask])
# count final state
for i, state in enumerate(state_names):
    state_mask = last_state_df['last_phone']==state
    transition_counts[i,-1]=len(last_state_df[state_mask])
transition_counts
```

[19]:
```
tensor([[  0.,   0.,   0.,   2.,   3.,   0.],
        [  0.,  75.,   0.,   0.,   1.,   1.],
        [  0.,   0., 103.,   1.,   1.,   1.],
        [  0.,   1.,   1.,  59.,   0.,   1.],
        [  0.,   1.,   2.,   0., 100.,   2.],
        [  0.,   0.,   0.,   0.,   0.,   0.]])
```

Notice how much bigger the counts are along the diagonal of the matrix, which correspond to instances of a state trainsitioning into itself. Also note the first column and last row are a bit useless: we never transition into the initial state nor out of the final state. Nevertheless, including

them in the matrix keeps the column and row indices consistent with what state they refer to.

Having calculated the transition counts, we need to transform these into probabilities by normalizing by the number of transitions out of each state. We can do this by dividing each row by the sum of all values on that row.

[20]:
```
transitions_out = transition_counts.sum(axis=1)
# reshape to column vector so we divide by rows
transitions_out=transitions_out.reshape((6,1))
transitions_out
```

[20]:
```
tensor([[  5.],
        [ 77.],
        [106.],
        [ 62.],
        [105.],
        [  0.]])
```

Again, since the bottom row is the final state there are no transitions out of it. To avoid dividing by zero, we'll just set the last element of `transitions_out` to 1, which won't hurt anything as, of course, 0/1 is still zero.

[21]:
```
transitions_out[-1]=1
transition_mat=transition_counts/transitions_out
print(transition_mat)
```

```
tensor([[0.0000, 0.0000, 0.0000, 0.4000, 0.6000, 0.0000],
        [0.0000, 0.9740, 0.0000, 0.0000, 0.0130, 0.0130],
        [0.0000, 0.0000, 0.9717, 0.0094, 0.0094, 0.0094],
        [0.0000, 0.0161, 0.0161, 0.9516, 0.0000, 0.0161],
        [0.0000, 0.0095, 0.0190, 0.0000, 0.9524, 0.0190],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000]])
```

To check we did our math right, let's make sure each row (except the last) now sums to 1.

[22]:
```
transition_mat.sum(axis=1)
```

[22]:
```
tensor([1., 1., 1., 1., 1., 0.])
```

We can now use this matrix to re-initialize the model edges. We can use the `start` attribute of the HMM to represent the start state. Recall that earlier we saved the distributions to a list named `states`.

[23]:
```
def add_hmm_edges(hmm, transition_mat, states):
    for i, state1 in enumerate([hmm.start,]+states):
        for j, state2 in enumerate(states+[hmm.end,], start=1):
            weight = transition_mat[i,j]
            if weight==0:
                continue
            if i==0 and j==len(states)+1:
```

```
                continue
            hmm.add_edge(state1, state2, weight)

add_hmm_edges(hmm, transition_mat, states)
hmm.edges, hmm.starts
```

[23]:
```
(tensor([[-0.0263, -1.3863, -1.3863, -4.3438],
         [-1.3863, -0.0287, -4.6634, -4.6634],
         [-4.1271, -4.1271, -0.0496, -1.3863],
         [-4.6540, -3.9608, -1.3863, -0.0488]]),
 tensor([   -inf,    -inf, -0.9163, -0.5108]))
```

Now that we've got our observation and transition probabilities taken care of, let's look at how to model the probability of an entire sequence.

6 Likelihood assignment

(Jurafsky & Martin 2009, Ch. 6, p. 179) Now that we've initialized the models parameters, let's look at how to get the probability for a given sequence of observations, $P(O|Q, \lambda)$. Recall that every observation depends on the state the produced it, $P(O|Q)$, and every state depends on the previous state, $P(q_t|q_{t-1})$. The latter probability we can simply get via lookup from the `transition_mat` object. To get the former, let's see how to get probability for a given observation from the distributions of the HMM using the `probability()` function.

[24]:
```
a_gmm = hmm.distributions[0]

# re-use samples generated earlier as k-means centroids
a_gmm.probability(sample_a).item(), a_gmm.probability(sample_n).item()
```

[24]:
```
(0.00053664471488446, 0.00010689154441934079)
```

Notice how small each number is. When modelling high-dimensional data, probabilities tend to get infinitesimally small, and can even be so small the computer is forced to round them down to zero. This problem is known as 'underflow', and is one of the reasons that log probability (straightforwardly, the log of the probability) is used more often than actual probability.

[25]:
```
a_gmm.log_probability(sample_a).item(), a_gmm.log_probability(sample_n).item()
```

[25]:
```
(-7.530174255371094, -9.143695831298828)
```

Unsurprisingly, `sample_a` has a much higher likelihood of being emitted by [a] than `sample_n` does. Conversely, if we score the probabilities of the same samples using the model for [n]:

[26]:
```
n_gmm = hmm.distributions[3]
n_gmm.log_probability(sample_n).item(), n_gmm.log_probability(sample_a).item()
```

[26]:
```
(-4.46790075302124, -93.80657958984375)
```

The probability that `sample_a` was emitted by [n] is even smaller.

Now let's consider the joint likelihood of O and a specific sequence of hidden states Q, i.e. $P(O, Q|\theta)$. We can model this as a product across all timesteps of the emission probability of the observation at that time by the hidden state with the transition probability of the hidden state from the previous hidden state. Let's use the states and observations for the word 'lawn' as an example.

Since the product of many probabilities quickly becomes infinitesimal (a problem known as 'underflow'), instead of multiplying probabilities we'll be adding log probabilities.

[27]:
```
lawn_mask = df['word']=='lawn'
word_observations = X[lawn_mask]
word_states = df.loc[lawn_mask, 'phone'].apply(state_names.index)
word_prev_states = df.loc[lawn_mask, 'last_phone'].apply(state_names.index)

# turn into log probs
transition_mat_log = (transition_mat+1e-10).log() # avoid taking log of 0

word_likelihood = 0
for observation, state_i, prev_state_i in zip(word_observations, word_states,␣
 ↪word_prev_states):
    state_distribution = hmm.distributions[state_i-1] # shift left since there␣
 ↪isn't any distribution for start
    emission_logprob = state_distribution.log_probability(observation.
 ↪reshape(1,2))
    transition_logprob = transition_mat_log[prev_state_i, state_i]
    word_likelihood+=emission_logprob+transition_logprob
word_likelihood.item()
```

[27]:
```
-327.7899169921875
```

Let's do the same again but with the same observations and a random sequence of states and see how the likelihood changes. Let's also encapsulate this into a function so we can use it later on.

[28]:
```
word_states_rand = np.random.choice(4, size=word_states.shape)

def joint_state_observation_prob(states, observations):
    log_likelihood = 0
    prev_states = np.insert(states, 0, 0)[:-1]
    for observation, state_i, prev_state_i in zip(observations, states,␣
 ↪prev_states):
        state_distribution = hmm.distributions[state_i-1] # since there isn't␣
 ↪any distribution for start
        emission_logprob = state_distribution.log_probability(observation.
 ↪unsqueeze(0))
        transition_logprob = transition_mat_log[prev_state_i, state_i]
        log_likelihood+=emission_logprob+transition_logprob
    return log_likelihood.item()
word_likelihood_rand = joint_state_observation_prob(word_states_rand,␣
 ↪word_observations)
```

```
word_likelihood_rand
```

[28]:
```
-5469.267578125
```

To see how big that difference is, exponentiate the difference in order to find the ratio of likelihoods, you should get `inf`!

[29]:
```
(word_likelihood-word_likelihood_rand).exp()
```

[29]:
```
tensor([inf])
```

Since we're considering the likelihood of the observations O regardless of the particular hidden state sequence, we have to consider **every hidden state** for each time step. To do this, we need to consider every possible permutation of hidden states across all timesteps. Let's start by just trying to compute this list of permutations without scoring any probabilities:

[30]:
```
from tqdm import tqdm

word = 'lawn'
word_features = df.loc[df['word']=='lawn', feat_cols].to_numpy()


def get_state_permutations(features):
    state_permutations = [[0]]
    for _ in tqdm(range(len(features))):
        timestep_paths = []
        for state_i in range(1, len(state_names)):
            for path in state_permutations:
                timestep_paths.append(path+[state_i,])
        state_permutations.extend(timestep_paths)
    return state_permutations
permutations = get_state_permutations(word_features[:10]) # try more iterations␣
 ↪if you dare
len(permutations)
```

```
100%|      | 10/10 [00:09<00:00,  1.08it/s]
```

[30]:
```
9765625
```

This has a time complexity of $O(n^T)$, with n being the number of states and T being the number of timesteps. Concretely, each iteration of the loop is n times as long as the previous. So, even for this relatively simple example, it takes at least $4^{75} = 142724769270595988105828596944949513638274662 4$(!!!) calculations to complete.

We can do a lot better using a simple dynamic programming trick: instead of computing the probability each path separately, recursively compute the probability of sub-paths and build off of each sub-paths probability. We do this by creating a *forward trellis*, which we represent with the matrix `forward_mat` containing one row for each state and one column for each timestep. Each cell represents the probability of **all possible subpaths** starting from the beginning and leading into that cell.

To compute this, we start by initializing each row (state) of the first column (i.e. first timestep) as the probability of transitioning into that state from the initial state multiplied by the emission probability of the initial observation for the given state.

Then, for each subsequent timestep we calculate the forward value for a given state as the sum for all previous states of the forward value of that state at the previous timestep multiplied by the transition probability of that state into the current state.

[31]:
```
# set matrices to float128 to prevent underflow
# (using numpy as pytorrhc doesnt support float128)
forward_mat = np.zeros((6,len(word_observations)), dtype=np.float128)
transition_mat_128 = transition_mat.numpy().astype(np.float128)

# initial timestep
initial_observation = word_observations[0]
for j, state in enumerate(states, start=1):
    transition_prob=transition_mat_128[0,j] # where 0 indicates the initial␣
 ↪state
    emission_prob=state.probability(initial_observation.unsqueeze(0)).item()
    forward_mat[j,0]=transition_prob*emission_prob

# remaining timesteps
for t, observation in enumerate(word_observations[1:], start=1):
    for j, curr_state in enumerate(states, start=1):
        emission_prob = curr_state.probability(observation.unsqueeze(0)).item()
        for i, _ in enumerate(states, start=1):
            transition_prob = transition_mat_128[i,j]
            prev_forward = forward_mat[i,t-1]
            forward_mat[j,t]+=prev_forward*emission_prob*transition_prob
        # print(forward_mat[:,t]) # uncomment to see how quickly numbers␣
 ↪underflow
# transitions to end state
for i, state in enumerate(states, start=1):
    transition_prob=transition_mat_128[i,-1] # where -1 indicates the final␣
 ↪state
    prev_forward=forward_mat[i,-1]
    forward_mat[-1,-1]+=prev_forward*transition_prob
word_prob=forward_mat[-1,-1]
word_prob
```

[31]:
```
3.1844271695947884166e-143
```

You should get a number with over **100 zeroes** after it. That's an absurdly small number. This happens because we're multiplying probabilities rather than adding log probabilities. Uncomment the print statement to visualize how quickly the probabilities underflow.

If we want to use logs, we'll have to change the code a bit, since in the lowermost for loop we're adding probabilities together, which is a problem since $\sum \log(p) \neq \log \sum p$. Instead, we'll have to exponentiate the log probabilities to convert them back into regular probabilities, sum and *then*

take the log. Unfortunately, because of this, we still need to use `np.float.128`, but now we only need it for that particular calculation, and we can store the `forward_mat` with doubles.

First, let's define a function that will exponentiate, add, and log an array of values for us to make our code a bit cleaner.

[32]:
```
def add_logprobs(log_probs: np.ndarray) -> float:
    if hasattr(log_probs, 'detach'):
        log_probs = log_probs.detach()
    # need quadruple precision to prevent underflow
    if log_probs.dtype is not np.float128:
        log_probs=np.array(log_probs, dtype=np.float128)
    probs=np.exp(log_probs)
    probs_sum=probs.sum()
    logprob_sum=np.log(probs_sum)
    return logprob_sum.astype(float)
```

Now let's define the `forward` function.

[33]:
```
def forward(observations, transition_mat_log, states):
    forward_mat = torch.full((len(states)+2,len(observations)), -torch.inf)
    # initial timestep
    forward_mat[0,0]=0 # always start in initial state
    initial_observation = observations[0]
    for j, state in enumerate(states, start=1):
        transition_logprob=transition_mat_log[0,j]
        emission_logprob=state.log_probability(initial_observation.
 ↪reshape([1,2]))
        forward_mat[j,0]=transition_logprob+emission_logprob
    # remaining timesteps
    for t, observation in enumerate(observations[1:], start=1):
        for j, curr_state in enumerate(states, start=1):
            emission_logprob = curr_state.log_probability(observation.
 ↪reshape([1,2])).item()
            logprobs = torch.zeros(len(states))
            for i, _ in enumerate(states, start=1):
                transition_logprob = transition_mat_log[i,j]
                prev_forward = forward_mat[i,t-1]
                logprobs[i-1]=prev_forward+transition_logprob
            logprob=add_logprobs(logprobs)
            logprob+=emission_logprob
            forward_mat[j,t]=logprob
    # transitions to end state
    end_logprobs = torch.zeros(len(states))
    for i, state in enumerate(states, start=1):
        transition_logprob=transition_mat_log[i,-1]
        prev_forward = forward_mat[i,-1]
        end_logprobs[i-1]=transition_logprob+prev_forward
```

```
    logprob=add_logprobs(end_logprobs)
    forward_mat[-1,-1]=logprob
    return logprob, forward_mat
logprob, forward_mat = forward(word_observations, transition_mat_log, states)
logprob
```

[33]: -328.11163330078125

Side note: in real-world implementations of the Forward and related sequence algorithms, underflow can be further addressed by scaling the probabilities during calculation. See this presentation and this post for more info.

As a sanity check, let's check that `pomegranate` gives us the same log probability for the sequence.

[34]:
```
hmm.log_probability(word_observations.unsqueeze(0)).item()
```

[34]: -327.5914001464844

Out of curiosity, let's see how the probability changes when we reverse the order of observations.

[35]:
```
logprob, forward_mat = forward(word_observations.flip(0), transition_mat_log,␣
↪states)
logprob
```

[35]: -343.62776004407567

Unsurprisingly, it's lower since we're traversing that states in the opposite direction to what the edge weights were tuned on.

7 Decoding

(Jurafsky & Martin 2009, Ch. 6, p. 184)

The next question to answer is how to get the optimal state sequence Q for producing an observation sequence O. We can do this in a similar way to how we computed overall likelihood, save that instead of aggregating the probability of each path, we instead return the maximum likely path. Instead of computing a Forward trellis, then, we compute a Viterbi trellis. Where each cell in the Forward trellis represents the cumulative likelihood of all paths leading to that cell, each cell in the Viterbi trellis represents the likelihood of the **single most likely path** leading to that cell.

In order to reconstruct the path, we also need to store a `backtrace` matrix. For a given state j at time t, this matrix stores the index of the most likely immediately preceding state i at time $t-1$ that led to that state. To calculate the most likely path, we use the backtrace to figure out what state was most likely to precede the final state, and what state was most likely to precede that, and so on through all timesteps.

One nice thing about this algorithm is that not worrying about cumulative probabilities means we don't need to convert log probabilities back into regular probabilities at any point, since at no point do we sum over several probabilities. Instead, we can work in log probability space the whole time.

A side effect of this is that we don't actually need to work with for-loops over states of the previous step. Since all we're doing is *adding* probabilities, we can just take the vector of previous states from the Viterbi matrix and add it to the vector of transition probabilities into the current state, then add the emmission probability to the resulting vector.

[36]:
```
reversed_enum = lambda a, start=0: reversed(list(enumerate(a, start=start)))

def viterbi(observations, transition_mat_log, states):
    viterbi_mat = torch.full((len(states)+2,len(observations)), fill_value=-np.
 ↪inf)
    backtrace = torch.zeros_like(viterbi_mat, dtype=int)

    # initial timestep - same as before
    initial_observation = observations[0]
    for j, state in enumerate(states, start=1):
        transition_logprob=transition_mat_log[0,j]
        emission_logprob=state.log_probability(initial_observation.
 ↪reshape([1,2]))
        viterbi_mat[j,0]=transition_logprob+emission_logprob

    # remaining timesteps
    for t, observation in enumerate(observations[1:], start=1):
        for j, curr_state in enumerate(states, start=1):
            emission_logprob = curr_state.log_probability(observation.
 ↪reshape([1,2]))
            prev_viterbi_vec = viterbi_mat[:,t-1]
            transition_vec = transition_mat_log[:,j]
            path_likelihoods = prev_viterbi_vec+transition_vec+emission_logprob

            max_path_likelihood = path_likelihoods.max()
            likely_prev_state = path_likelihoods.argmax() # argmax returns the␣
 ↪index of the max value

            viterbi_mat[j,t]=max_path_likelihood
            backtrace[j,t]=likely_prev_state

    # transitions to end state
    final_viterbi_vec = viterbi_mat[:,-1]
    final_transition_vec = transition_mat_log[:,-1]
    final_likelihoods = final_viterbi_vec + final_transition_vec
    max_final_likelihood = final_likelihoods.max()
    likely_prefinal_state = final_likelihoods.argmax()

    viterbi_mat[-1,-1]=max_final_likelihood
    backtrace[-1,-1]=likely_prefinal_state

    # decode path from backtrace
```

```
    prev_state = likely_prefinal_state
    path = torch.zeros(len(observations+2), dtype=int)
    path[-1]=-1
    # so we can iterate thru columns
    backtrace_iter = backtrace.transpose(0,1)
    for t, idcs in reversed_enum(backtrace_iter):
        path[t]=prev_state
        prev_state=idcs[prev_state]
    return path, viterbi_mat
path, viterbi_mat = viterbi(word_observations, transition_mat_log, states)
path
```

[36]:
```
tensor([3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
        4, 4, 4])
```

If we do a quick substitution of state indices with there string representation, we can see that we've correctly predicted the sequence [lan] for 'lawn'.

[37]:
```
''.join([state_names[i] for i in path])
```

[37]:
```
'lllllllllaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaannnnnnnnnnnnnnnnnnnnnnn'
```

Let's compare our output with the output of Pomegranate's `viterbi` function, and also with the ground truth from the Praat textgrids.

[38]:
```
for word in words:
    word_mask = df['word']==word
    word_X = X[word_mask]
    word_Y = df.loc[word_mask,'phone'].tolist()
    print(word)
    pomegranate_preds = hmm.viterbi(np.reshape(word_X, (1,-1,2))).squeeze()
    # add 1 since 0th state for pomegranate is [a], not start
    pomegranate_decoded = ''.join([state_names[i+1] for i in pomegranate_preds])
    print('Pomegranate viterbi:\t',pomegranate_decoded)

    viterbi_preds, _ = viterbi(word_X, transition_mat_log, states)
    viterbi_decoded = ''.join([state_names[i] for i in viterbi_preds])
    print('Our viterbi:\t\t', viterbi_decoded)

    print('Ground truth:\t\t', ''.join(word_Y))
```

```
lawn
Pomegranate viterbi:
lllllllllaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaannnnnnnnnnnnnnnnnnnnnnn
Our viterbi:
lllllllllaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaannnnnnnnnnnnnnnnnnnnnnn
Ground truth:
```

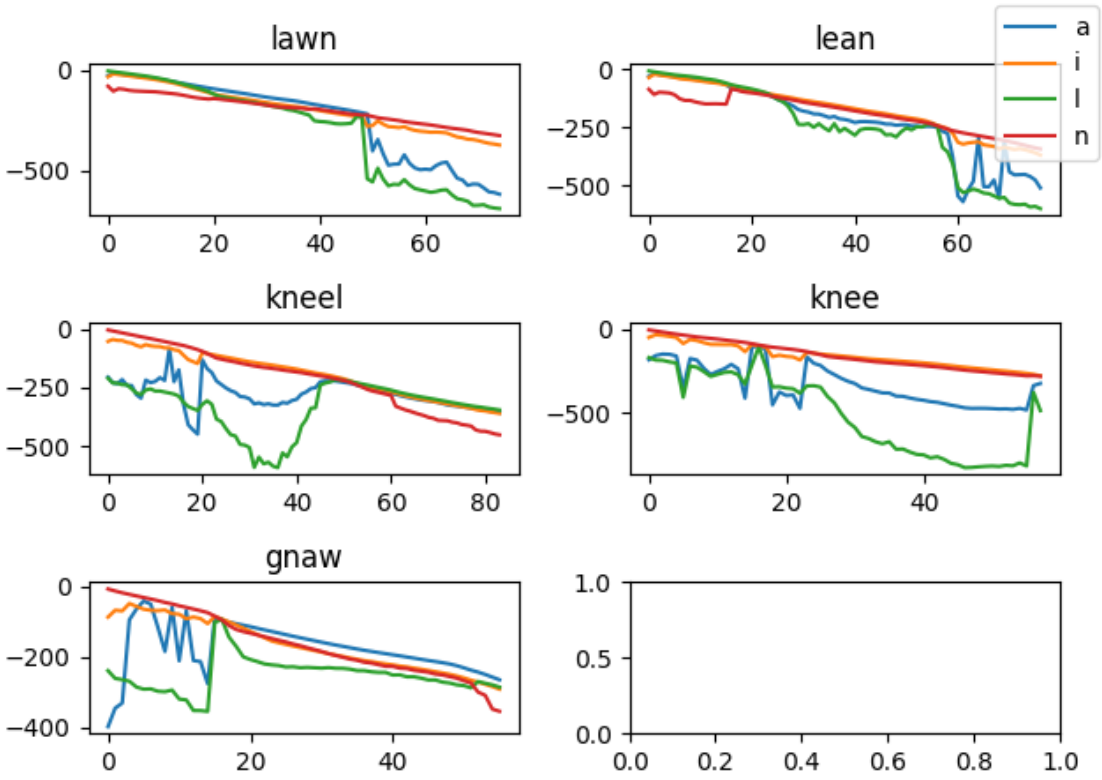
```
lllllllllllllaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaannnnnnnnnnnnnnnnnnnnnnnnn
lean
Pomegranate viterbi:
llllllllllllllllllllllliiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiinnnnnnnnnnnnnnnnnnnnnn
Our viterbi:
llllllllllllllllllllllliiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiinnnnnnnnnnnnnnnnnnnnnn
Ground truth:
llllllllllllllliiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiinnnnnnnnnnnnnnnnnn
kneel
Pomegranate viterbi:     nnnnnnnnnnnnnnnnnnnnniiiiiiiiiiiiiiiiiiiiiiiiiiiiiillll
lllllllllllllllllllllllllllll
Our viterbi:             nnnnnnnnnnnnnnnnnnnnniiiiiiiiiiiiiiiiiiiiiiiiiiiiiillll
lllllllllllllllllllllllllllll
Ground truth:            nnnnnnnnnnnnnnnnnnnnniiiiiiiiiiiiiiiiiiiiiiiiiiiiilllll
lllllllllllllllllllllllllllll
knee
Pomegranate viterbi:
nnnnnnnnnnnnnnnnnnnnnnnniiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
Our viterbi:
nnnnnnnnnnnnnnnnnnnnnnnniiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
Ground truth:
nnnnnnnnnnnnnnnnnnnnnnnnniiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
gnaw
Pomegranate viterbi:
nnnnnnnnnnnnnnnllaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaall
Our viterbi:
nnnnnnnnnnnnnnnaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Ground truth:
nnnnnnnnnnnnnnnnaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Pretty similar! There most unexpected difference (when I ran it) is the insertion of some anomalous [l]‘s before and after [a] in the Pomegranate output for ’gnaw.’

Let’s use the Viterbi matrix to visualize the relative probability of each state over the course of the word.

[39]:
```python
fig, axes = plt.subplots(nrows=3,ncols=2)
flat_axes=axes.flatten()
for i,word in enumerate(words):
    word_mask = df['word']==word
    word_X = X[word_mask]
    word_Y = df.loc[word_mask,'phone'].tolist()
    _, viterbi_mat = viterbi(word_X, transition_mat_log, states)
    plot_viterbi = viterbi_mat.transpose(0,1)[:,1:-1]
    flat_axes[i].plot(plot_viterbi, label=[*state_names[1:]])
    flat_axes[i].set_title(word)

handles, labels = flat_axes[0].get_legend_handles_labels()
```

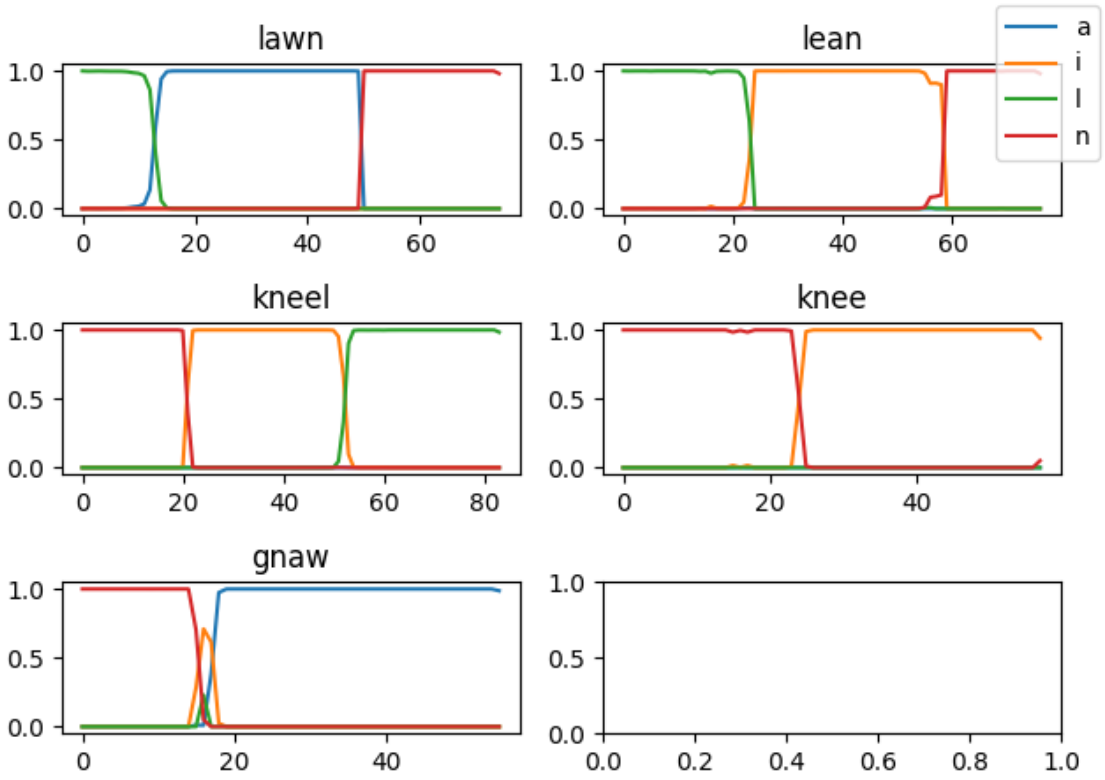
```
fig.legend(handles, labels, loc='upper right')
plt.tight_layout(rect=[0, 0.05, 1, 1])
plt.show()
```



Well that's a little hard to interpret. Naturally, all of the probabilities decrease over time, as the longer the sequence becomes the lower the overall probabilities. One line crossing another indicates a transition between phones. To make this more interpretable, we can compute a softmax over each timestep to get relative probabilities for each phone rather than overall likelihood of the sequence.

[40]:
```
from scipy.special import softmax

def plot_viterbi(states=None, transition_mat=None, hmm_dict=None,␣
 ↪transition_mat_dict=None):
    fig, axes = plt.subplots(nrows=3,ncols=2)
    flat_axes=axes.flatten()
    for i, word in enumerate(words):
        if transition_mat_dict:
            transition_mat = transition_mat_dict[word]

        if hmm_dict:
            states = hmm_dict[word].distributions
        word_mask = df['word']==word
```

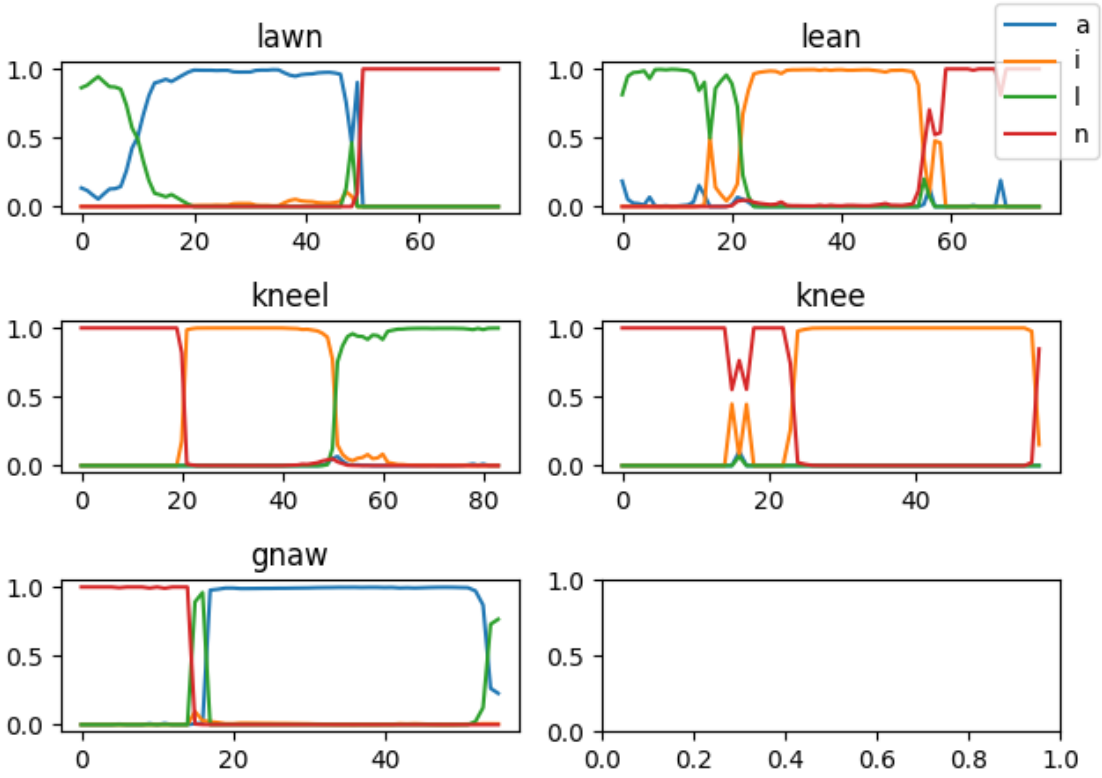
```
        word_X = X[word_mask]
        path, viterbi_mat = viterbi(word_X, transition_mat, states)
        viterbi_mat_softmax = softmax(viterbi_mat, axis=0)
        viterbi_plot_data = viterbi_mat_softmax.transpose()[:,1:-1]
        flat_axes[i].plot(viterbi_plot_data, label=[*state_names[1:]])
        flat_axes[i].set_title(word)

    handles, labels = flat_axes[0].get_legend_handles_labels()
    fig.legend(handles, labels, loc='upper right')
    plt.tight_layout(rect=[0, 0.05, 1, 1])
    plt.show()
plot_viterbi(states, transition_mat=transition_mat_log)
```



Much easier to interpret. Now we can see the transition between phones very clearly. How does this compare to a plot of probabilities using just GMMs, no state transitions from HMMs?

[41]:
```
fig, axes = plt.subplots(nrows=3,ncols=2)
flat_axes=axes.flatten()
for i, word in enumerate(words):
    word_mask = df['word']==word
    word_X = X[word_mask]
    word_Y = df.loc[word_mask,'phone'].tolist()
```

```
    gmm_likelihoods = torch.zeros((len(word_X),len(states)))
    for j,state in enumerate(states):
        state_prob = state.log_probability(word_X)
        gmm_likelihoods[:,j]=state_prob
    gmm_softmax = softmax(gmm_likelihoods, axis=1)
    flat_axes[i].plot(gmm_softmax, label=[*state_names[1:]])
    flat_axes[i].set_title(word)

handles, labels = flat_axes[0].get_legend_handles_labels()
fig.legend(handles, labels, loc='upper right')
plt.tight_layout(rect=[0, 0.05, 1, 1])
plt.show()
```



Pretty similar overall, though the GMM plots are a bit noisier, as can be seen in the dip in [n] probability in the onset for 'knee' or the fact that [a] and [l] cross over each other at the end of 'gnaw', as if the word were [nal]. This shows that the HMM is doing some work to smooth over the phone transitions.

8 Fitting HMM-Gaussian

(Jurafsky & Martin 2009, Ch. 6, p. 187; Ch. 9 p. 308)

To keep things simple, we're going to stick with non-mixed Gaussians for training our toy Forced Aligner, hence the section title "HMM-Gaussian" rather than "HMM-GMM".

By now we've discussed how to use an HMM to for likelihood estimation and decoding, however we dealt with an HMM that was fit to hand-segmented data. Ideally, though, we'd like to be able to use an HMM without having to hand-segment the data we run it on. To this end, we can use unsupervised training with Expectation Maximization (EM).

The general idea behind EM for HMM-GMM training is this: The parameters of the HMM-GMM should ideally represent the distribution of phone and acoustic sequences we observe in the real world. Using a dataset of speech, we can calculate the average number of transitions between states and emissions of acoustic observations by states that the model predicts for the speech data. If we represent the parameters for transition and emission probability as a and b respectively, we can say that $\hat{a}$ and $\hat{b}$ are the *expected number of transitions and emissions* given the dataset. These latter values are likely to be closer to the ground truth distribution of transitions and emissions than the original model parameters, so we can substitute $\hat{a}$ and $\hat{b}$ in for a and b, then repeat the process again until we've converged. For more reading on training with EM (beyond the Jurafsky & Martin textbook), see this medium article on HMM-GMM for speech recognition and medium article for Expectation Maximization

First, since we're not assuming any presegmented data, let's fit each Gaussian model to the entire dataset of acoustic observations (that is, each phone state will actually be modeling all phones at once!)

[42]:
```
bw_states = [Normal().fit(X) for _ in range(4)]
```

For the transition states, we'll actually create a different matrix for each word. After all, though we don't know the exact transition probabilities, we do know that for a given word, only a subset of transitions are possible. Namely, a given phone can either transition into itself or into the following state. It cannot skip a phone, nor can it transition to a previous phone. We can model this by setting for each state an equal probability of transitioning to itself or to the next phone in the sequence. For the initial state, we simply give a probability of 1 that it will transition into the first phone state of the word.

[43]:
```
# state_names = '^ailn'
# words = ['lawn', 'lean', 'kneel', 'knee', 'gnaw']
words_ipa = df['word_ipa'].unique()
word_transitions = {}

for word, word_ipa in zip(words, words_ipa):
    word_trans_mat = torch.zeros((6,6))
    # only one possible transition from initial state
    word_trans_mat[0,state_names.index(word_ipa[0])]=1
    for i, char in enumerate(word_ipa):
        char_i = state_names.index(char)
        if i < len(word_ipa)-1:
            next_char = word_ipa[i+1]
            next_char_i = state_names.index(next_char)
        else:
```

```
            next_char = '$'
            next_char_i = -1
        # equal likelihood transition to self or next char
        word_trans_mat[char_i, next_char_i] = 0.5
        word_trans_mat[char_i, char_i] = 0.5
    word_transitions[word]=word_trans_mat

word_transitions
```

[43]:
```
{'lawn': tensor([[0.0000, 0.0000, 0.0000, 1.0000, 0.0000, 0.0000],
         [0.0000, 0.5000, 0.0000, 0.0000, 0.5000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.5000, 0.0000, 0.5000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.5000, 0.5000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000]]),
 'lean': tensor([[0.0000, 0.0000, 0.0000, 1.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.5000, 0.0000, 0.5000, 0.0000],
         [0.0000, 0.0000, 0.5000, 0.5000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.5000, 0.5000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000]]),
 'kneel': tensor([[0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.5000, 0.5000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.5000, 0.0000, 0.5000],
         [0.0000, 0.0000, 0.5000, 0.0000, 0.5000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000]]),
 'knee': tensor([[0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.5000, 0.0000, 0.0000, 0.5000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.5000, 0.0000, 0.5000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000]]),
 'gnaw': tensor([[0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 0.0000],
         [0.0000, 0.5000, 0.0000, 0.0000, 0.0000, 0.5000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.5000, 0.0000, 0.0000, 0.5000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000]])}
```

Now let's take the log of these matrices.

[44]:
```
word_transitions_log = {k:v.log() for k,v in word_transitions.items()}
word_transitions_log
```

[44]:
```
{'lawn': tensor([[   -inf,    -inf,    -inf,  0.0000,    -inf,    -inf],
         [   -inf, -0.6931,    -inf,    -inf, -0.6931,    -inf],
         [   -inf,    -inf,    -inf,    -inf,    -inf,    -inf],
```

```
        [   -inf, -0.6931,    -inf, -0.6931,    -inf,    -inf],
        [   -inf,    -inf,    -inf,    -inf, -0.6931, -0.6931],
        [   -inf,    -inf,    -inf,    -inf,    -inf,    -inf]]),
 'lean': tensor([[   -inf,    -inf,    -inf,  0.0000,    -inf,    -inf],
        [   -inf,    -inf,    -inf,    -inf,    -inf,    -inf],
        [   -inf,    -inf, -0.6931,    -inf, -0.6931,    -inf],
        [   -inf,    -inf, -0.6931, -0.6931,    -inf,    -inf],
        [   -inf,    -inf,    -inf,    -inf, -0.6931, -0.6931],
        [   -inf,    -inf,    -inf,    -inf,    -inf,    -inf]]),
 'kneel': tensor([[   -inf,    -inf,    -inf,    -inf,  0.0000,    -inf],
        [   -inf,    -inf,    -inf,    -inf,    -inf,    -inf],
        [   -inf,    -inf, -0.6931, -0.6931,    -inf,    -inf],
        [   -inf,    -inf,    -inf, -0.6931,    -inf, -0.6931],
        [   -inf,    -inf, -0.6931,    -inf, -0.6931,    -inf],
        [   -inf,    -inf,    -inf,    -inf,    -inf,    -inf]]),
 'knee': tensor([[   -inf,    -inf,    -inf,    -inf,  0.0000,    -inf],
        [   -inf,    -inf,    -inf,    -inf,    -inf,    -inf],
        [   -inf,    -inf, -0.6931,    -inf,    -inf, -0.6931],
        [   -inf,    -inf,    -inf,    -inf,    -inf,    -inf],
        [   -inf,    -inf, -0.6931,    -inf, -0.6931,    -inf],
        [   -inf,    -inf,    -inf,    -inf,    -inf,    -inf]]),
 'gnaw': tensor([[   -inf,    -inf,    -inf,    -inf,  0.0000,    -inf],
        [   -inf, -0.6931,    -inf,    -inf,    -inf, -0.6931],
        [   -inf,    -inf,    -inf,    -inf,    -inf,    -inf],
        [   -inf,    -inf,    -inf,    -inf,    -inf,    -inf],
        [   -inf, -0.6931,    -inf,    -inf, -0.6931,    -inf],
        [   -inf,    -inf,    -inf,    -inf,    -inf,    -inf]])}
```

Next we initialize a different HMM for each word using the respective transition probabilities, with each HMM having the same set of Gaussian distributions for modeling observations as the other HMMs.

[45]:
```
word_hmms = {}

for word in words:
    word_hmm = DenseHMM()
    word_hmm.add_distributions(bw_states)
    add_hmm_edges(word_hmm, word_transitions[word], bw_states)
    word_hmms[word]=word_hmm
word_hmms['lawn'].edges, word_hmms['lawn'].starts
```

[45]:
```
(tensor([[-0.6931,    -inf,    -inf, -0.6931],
        [   -inf,    -inf,    -inf,    -inf],
        [-0.6931,    -inf, -0.6931,    -inf],
        [   -inf,    -inf,    -inf, -0.6931]]),
 tensor([-inf, -inf, 0., -inf]))
```

Now that we've initialized our models, let's go about estimating our new parameters for transition

and observation probabilities. To estimate the expected transition or emission probabilities over the whole dataset, we'll want to first be able to estimate the expected transition or emission probability *for a particular state at a particular point in time*, and then build up the average expected value from there.

Earlier, we calculated the *forward probability* for a state and time in a given sequence, represented as $\alpha_t(i)$. This is equivalent to the probability of all paths leading into state i at time t. This is not, however, the *absolute probability* of being in state i at time t. To calculate that, we need to consider the probability of all paths leading out from state i at time t going into the final state. This is the *backward probability*, $\beta_t(i)$.

To calculate the backward probability, we essentially reverse the forward algorithm: we start at the final timestep and set $\beta_T(i)$ as the emission probability of observation o_T by state i times the probability of state i transitioning into the final state q_F. Then for each preceding timestep t, sum up the product of the backwards probability for time $t+1$ for all states with the probability of i transitioning into each respective state times the emission probability of o_t by state i until we reach the beginning of the sequence. The value stored for the initial state at $t = 0$ in the backward matrix is equivalent to the overall sequence probability, just like the value stored for the final state at the final timestep in the forward matrix was the overall sequence probability.

[46]:
```
def backward(observations, transition_mat_log, states):
    backward_mat = torch.full((len(states)+2,len(observations)), -torch.inf)
    # final timestep
    backward_mat[-1,-1]=0 # always end in final state
    final_observation = observations[-1]
    for i, state in enumerate(states, start=1):
        transition_logprob=transition_mat_log[i,-1]
        emission_logprob=state.log_probability(final_observation.reshape([1,2]))
        backward_mat[i,-1]=transition_logprob+emission_logprob
    # remaining timesteps
    for t, observation in reversed_enum(observations[:-1]):
        for i, curr_state in enumerate(states, start=1):
            emission_logprob = curr_state.log_probability(observation.
 ↪reshape([1,2])).item()
            logprobs = torch.zeros(len(states))
            for j, _ in enumerate(states, start=1):
                transition_logprob = transition_mat_log[i,j]
                next_backward = backward_mat[j,t+1]
                logprobs[j-1]=next_backward+transition_logprob
            logprob=add_logprobs(logprobs)
            logprob+=emission_logprob
            backward_mat[i,t]=logprob
    # transitions to initial state
    init_logprobs = torch.zeros(len(states))
    for i, state in enumerate(states, start=1):
        transition_logprob=transition_mat_log[0,i]
        next_backward = backward_mat[i,0]
        init_logprobs[i-1]=transition_logprob+next_backward
```

```
    logprob=add_logprobs(init_logprobs)
    backward_mat[0,0]=logprob
    return logprob, backward_mat
back_logprob, backward_mat = backward(word_observations, transition_mat_log,␣
 ↪states)
back_logprob
```

[46]: -328.11138916015625

As a sanity check, let's make sure we get (roughly) the same value for the forward algorithm on the same sequence.

[47]:
```
forward_logprob, forward_mat = forward(word_observations, transition_mat_log,␣
 ↪states)
forward_logprob
```

[47]: -328.11163330078125

Now that we have both forward and backward probabilities available, we can now compute $\xi_t(i,j)$ and $\gamma_t(i)$. $\xi_t(i,j)$ is the probability that from time t and $t+1$, we transition from state i to j.

Recall that the forward and backward variables store the probability over **all possible subpaths** up to time t in their given direction. For $\xi_t(i,j)$, we are concerned with one specific path between timesteps t and $t+1$ (the path from i to j), but for all other timesteps we are not considering any one path in particular. For this reason, we can model $\xi_t(i,j)$ as:

$$\xi_t(i,j) = \frac{\alpha_t(i)\beta_{t+1}(j)a_{ij}b_i(o_t)}{\alpha_T(q_F)}$$

Where $\alpha_t(i)\beta_{t+1}(j)$ can be thought of as the probability of all paths leading to the transition between i and j in question, $a_{ij}b_i(o_t+1)$ is the probability of the transition itself, and $\alpha_T(q_F)$ is the overall probability of the entire sequence. Note we wouldn't want to consider $b_i(t)$ since the forward variable $\alpha_t(i)$ already includes this probability in its definition. By dividing by $\alpha_T(q_F)$, we consider the probability of the transition given the sequence it takes place in, rather than the absolute probability the HMM would predict.

[48]:
```
def ksi(i, j, t, observations, forward, backward, transition_mat_log, states):
    """
    i and j in [start, *states, end]
    """
    forward_i = forward[i,t]
    backward_j = backward[j,t+1]
    transition = transition_mat_log[i,j]
    if (j==len(states)+1) or (j==-1):
        # can't transition to final state before final timestep
        if t<len(observations)-1:
            return float('-inf')
        # else emission probability is 1 (log(1)=0) when transitioning to final␣
 ↪state at final timestep
        emission = 0
```

```
    else:
        emission = states[j-1].log_probability(observations[t+1].
↪reshape([1,2])).item()

    seq_prob = forward[-1,-1]
    if seq_prob == float('-inf'):
        return float('-inf')

    ksi_val = forward_i + backward_j + transition + emission - seq_prob
    return ksi_val

ksi(
    3,
    2,
    len(word_observations)-15,
    word_observations,
    forward(word_observations, transition_mat_log, states)[1],
    backward(word_observations, transition_mat_log, states)[1],
    transition_mat_log=transition_mat_log,
    states=states,
)
```

[48]:
```
tensor(-406.9579)
```

In addition to $\xi_t(i,j)$, we also compute $\gamma_t(i)$. This is equivalent to the probability of being in state i at time t and emitting observation o_t. Since $\alpha_t(i)$ is the probability of all paths leading *into* state i at time t, and $\beta_t(i)$ is the probability of all paths going *out of* state i at time t, we can calculate $\gamma_t(i)$ as:

$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\alpha_T(q_F)}$

Where $\alpha_T(q_F)$ again scales the probability relative to the overall probability of the sequence.

[49]:
```
def gamma(i, t, forward, backward):
    """
    i in [start, *states, end]
    """
    forward_i = forward[i,t]
    backward_i = backward[i,t]
    seq_prob = forward[-1,-1]
    return forward_i + backward_i - seq_prob

gamma(
    3,
    len(word_observations)-15,
    forward(word_observations, transition_mat_log, states)[1],
    backward(word_observations, transition_mat_log, states)[1],
)
```

[49]:
```
tensor(-664.5364)
```

[50]:
```
def a_hat(i, observations, transition_mat_log, states):
    """
    i and j in [start, *states, end]
    """

    ksi_sums = torch.full((len(states),), float('-inf'))
    _, forward_mat = forward(observations, transition_mat_log, states)
    _, backward_mat = backward(observations, transition_mat_log, states)

    for j in range(1, len(states)+1):
        ksi_j = torch.full((len(observations),), float('-inf'))
        for t in range(len(observations)-1):
            ksi_log =␣
 ↪ksi(i,j,t,observations,forward_mat,backward_mat,transition_mat_log,states)
            ksi_j[t]=ksi_log
        ksi_sums[j-1]=add_logprobs(ksi_j)

    total_ksi = add_logprobs(ksi_sums)
    if total_ksi == float('-inf'):
        return torch.full((len(states),), float('-inf'))
    a_hat_vec = ksi_sums - total_ksi
    return a_hat_vec

a_hat(1,word_observations,word_transitions_log['lawn'],states)
```

```
/var/folders/bt/_dsrh6ld2yncbnn9vk_d2lcc0000gp/T/ipykernel_99367/1710079399.py:9
: RuntimeWarning: divide by zero encountered in log
  logprob_sum=np.log(probs_sum)
```

[50]:
```
tensor([-3.5322e-04,        -inf,        -inf, -7.9486e+00])
```

[51]:
```
def mu_sigsq_hat(i, observations, transition_mat_log, states):
    """
    i in [start, *states, end]
    """
    _, forward_mat = forward(observations, transition_mat_log, states)
    _, backward_mat = backward(observations, transition_mat_log, states)

    # convert to numpy since we'll be using float128
    observations = observations.numpy()
    gamma_vec_log = np.array(
        [gamma(i,t,forward_mat,backward_mat) for t in range(len(observations))],
        dtype=np.float128,
    )
    gamma_vec = np.exp(gamma_vec_log)
    weighted_observations = observations*gamma_vec[:,None]
```

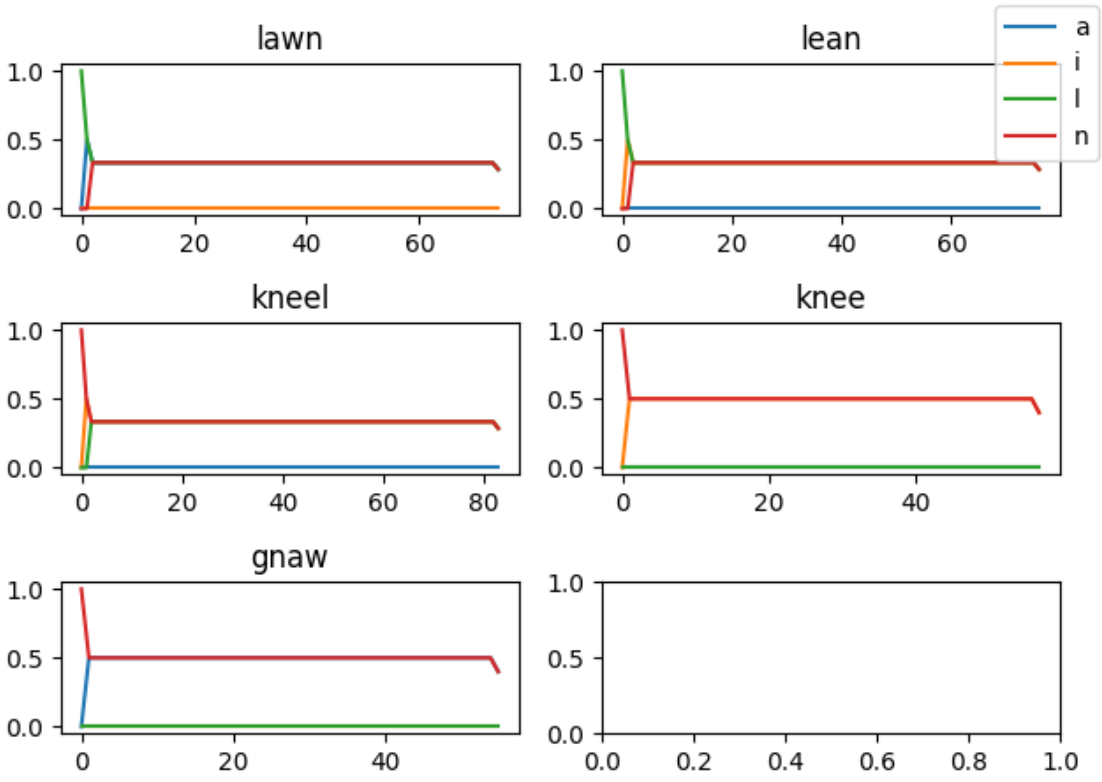
```
    mu_hat = weighted_observations.sum(axis=0)/gamma_vec.sum()

    observation_minus_mean = observations-mu_hat
    observation_minus_mean_dot = np.stack([column[:,None]@column[None,:] for␣
 ↪column in observation_minus_mean])
    numerator = observation_minus_mean_dot * gamma_vec[:,None,None]
    sigma_hat = numerator.sum(axis=0)/gamma_vec.sum()

    sigma_hat = torch.tensor(sigma_hat.astype(np.float64))
    mu_hat = torch.tensor(mu_hat.astype(np.float64))

    return mu_hat, sigma_hat

mu_sigsq_hat(1, word_observations, transition_mat_log, states)
```

[51]:
```
(tensor([-12.3937,   5.2621], dtype=torch.float64),
 tensor([[20.7811, -9.6813],
         [-9.6813,  6.1762]], dtype=torch.float64))
```

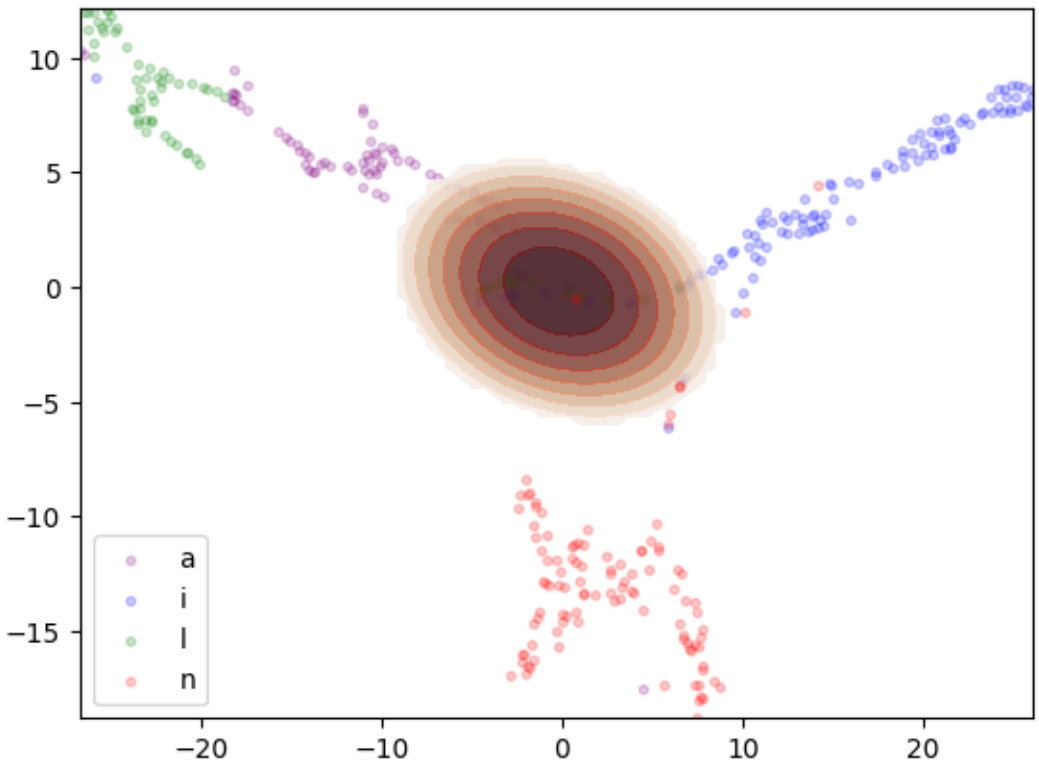
[52]:
```
def em_step(df, X, hmm_dict, word_transitions_dict, phones):
    num_states = len(list(hmm_dict.values())[0].distributions)
    state_means = torch.zeros((num_states, 2))
    state_covs = torch.zeros((num_states, 2, 2))
    new_transitions={}
    for word in df['word'].unique():
        word_mask = df['word']==word
        word_ipa = df.loc[word_mask, 'word_ipa'].iloc[0]
        state_idcs = list(set(phones.index(c)+1 for c in word_ipa))
        word_feats = X[word_mask]
        word_hmm = hmm_dict[word]
        states = word_hmm.distributions
        word_trans_mat = word_transitions_dict[word]

        new_transition_mat = torch.full_like(word_trans_mat, -torch.inf)
        new_transition_mat[0] = word_trans_mat[0] # initial transition␣
 ↪probabilities don't change
        for i in state_idcs:
            # expected transition probabilities
            a_hat_vec = a_hat(i, word_feats, word_trans_mat, states)
            print(word, i, a_hat_vec)
            # set transition probs for state i for given word
            new_transition_mat[i,1:-1]=a_hat_vec

            # collect emission probabilities
            mu_hat_vec, sigmasq_hat_mat = mu_sigsq_hat(i, word_feats,␣
 ↪word_trans_mat, states)
            weight_for_avg = len(word_feats)/len(df)
```

```
            state_means[i-1]+=mu_hat_vec*weight_for_avg
            state_covs[i-1]+=sigmasq_hat_mat*weight_for_avg
        print(new_transition_mat)
        add_hmm_edges(word_hmm, torch.exp(new_transition_mat), states)
        new_transitions[word]=new_transition_mat
    for i in range(1,num_states):
        states[i-1].means=torch.nn.Parameter(torch.tensor(state_means[i-1]),␣
↪requires_grad=False)
        states[i-1].covs=torch.nn.Parameter(torch.tensor(state_covs[i-1]),␣
↪requires_grad=False)
    return new_transitions, states
```

[53]:
```
plot_viterbi(hmm_dict=word_hmms, transition_mat_dict=word_transitions_log)
```

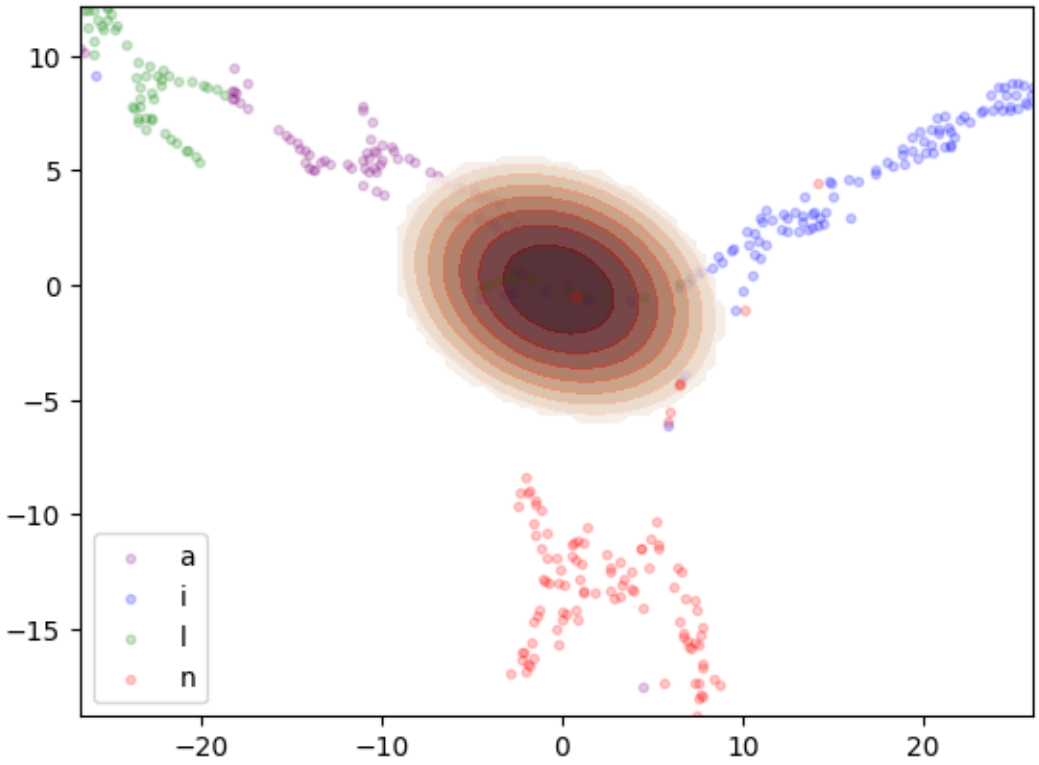


[54]:
```
plot_gaussians(bw_states)
```



[ ]:
```
trained_transitions = word_transitions_log
for i in range(5):
    trained_transitions, trained_states=em_step(df, X, word_hmms,␣
 ↪trained_transitions, phones)
plot_viterbi(hmm_dict=word_hmms, transition_mat_dict=trained_transitions)
```

[56]:
```
plot_gaussians(trained_states)
```



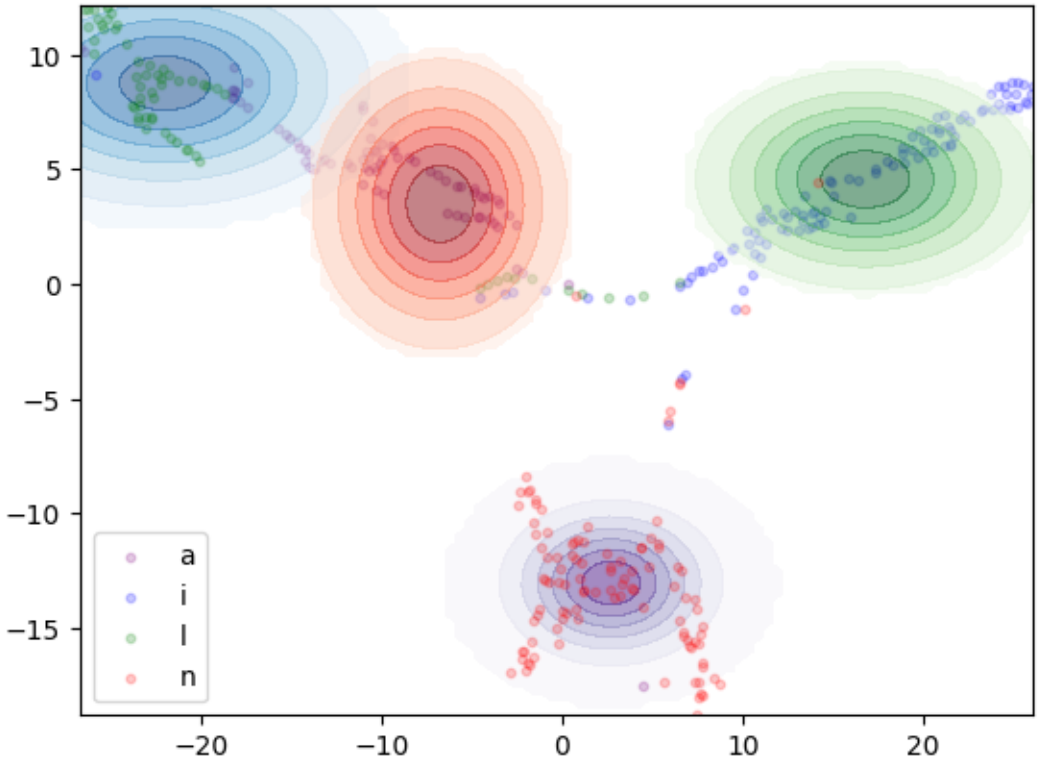
Seed Gaussians from k-means clusters

[57]:
```
cluster_means = torch.tensor(kmeans.cluster_centers_)
cluster_vars = torch.zeros(4,2)
seeded_states = []
for i, _ in enumerate(phones):
    cluster_mask = y_hat==i
    cluster_points = X[cluster_mask]
    var = X[cluster_mask].var(dim=0)
    cluster_vars[i]=var
    seeded_states.append(Normal(
        means=cluster_means[i],
        covs=var,
        covariance_type='diag'
    ))
seeded_states, cluster_means, cluster_vars
```

[57]:
```
([Normal(), Normal(), Normal(), Normal()],
 tensor([[  2.7162, -13.0293],
         [-21.9970,   8.7712],
         [ 16.7898,   4.5764],
         [ -6.7494,   3.5071]], dtype=torch.float64),
```

```
tensor([[ 9.9449,  3.3919],
        [34.1531,  7.5984],
        [22.8406,  5.9092],
        [12.9050, 10.3579]]))
```

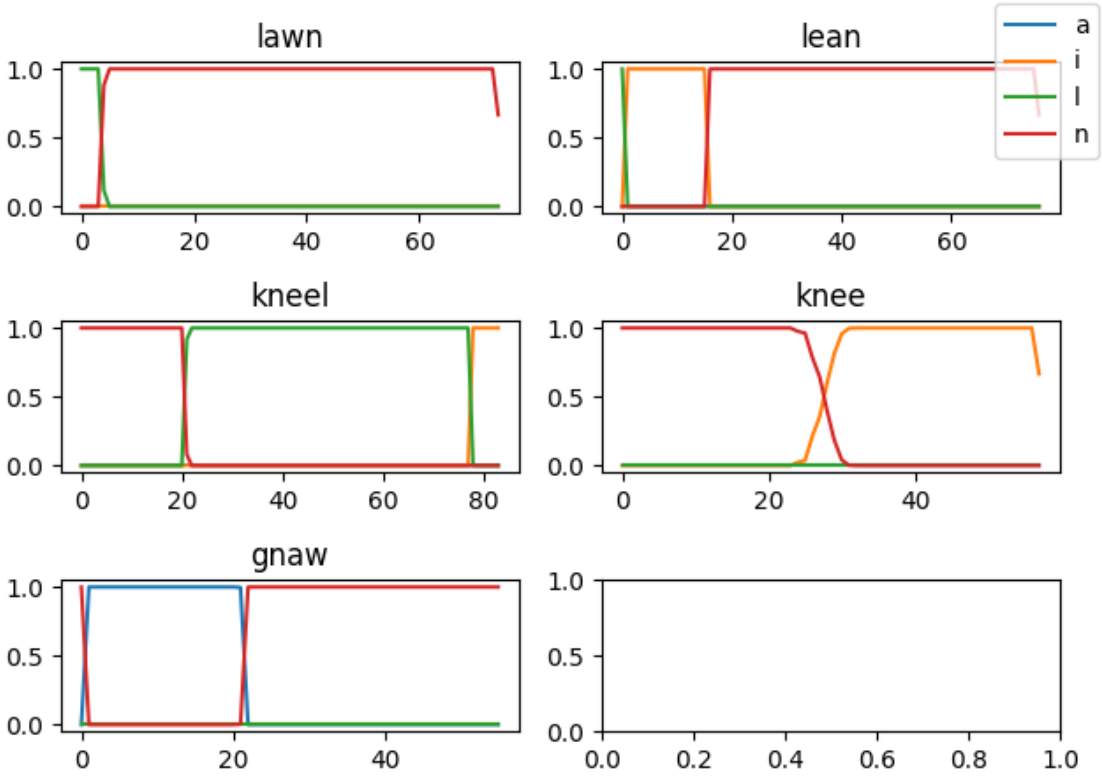
[58]:
```
plot_gaussians(seeded_states)
```



[59]:
```
seeded_hmms = {}

for word in words:
    word_hmm = DenseHMM()
    word_hmm.add_distributions(seeded_states)
    add_hmm_edges(word_hmm, word_transitions[word], seeded_states)
    seeded_hmms[word]=word_hmm
seeded_hmms['lawn'].edges, seeded_hmms['lawn'].starts
```

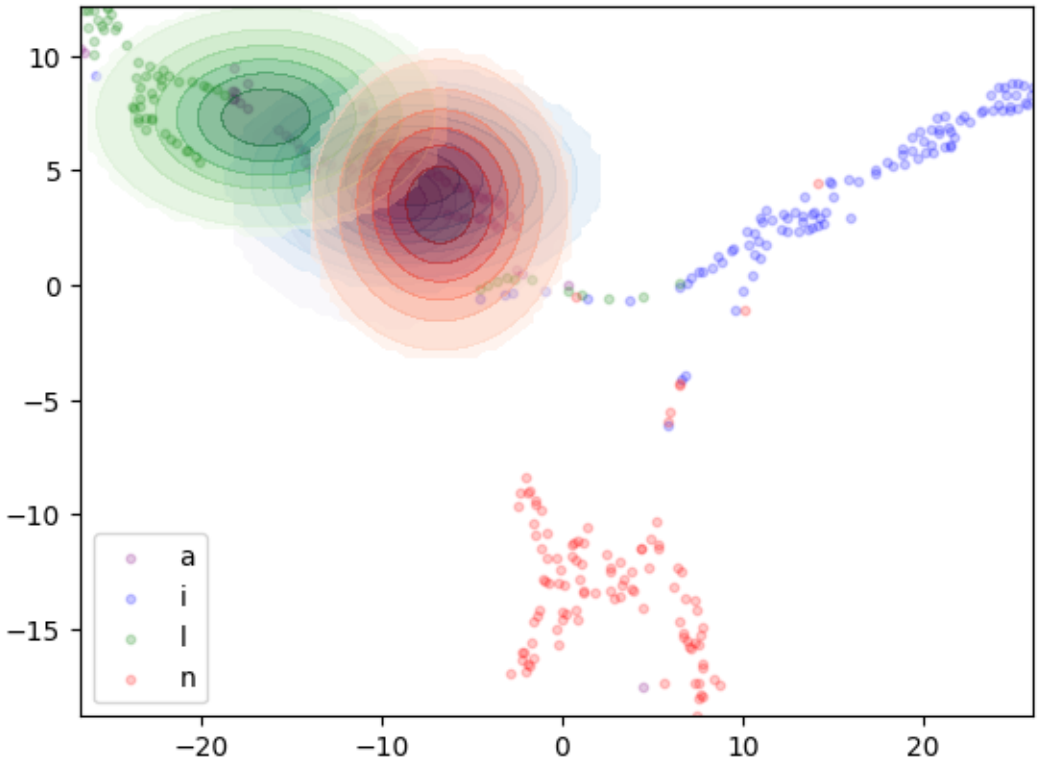
[59]:
```
(tensor([[-0.6931,    -inf,    -inf, -0.6931],
         [   -inf,    -inf,    -inf,    -inf],
         [-0.6931,    -inf, -0.6931,    -inf],
         [   -inf,    -inf,    -inf, -0.6931]]),
 tensor([-inf, -inf, 0., -inf]))
```

[60]:
```
plot_viterbi(hmm_dict=seeded_hmms, transition_mat_dict=word_transitions_log)
```



[ ]:
```
trained_transitions = word_transitions_log
for i in range(1):
    trained_transitions,trained_states=em_step(df, X, seeded_hmms,␣
 ↪trained_transitions, phones)
plot_viterbi(hmm_dict=seeded_hmms, transition_mat_dict=trained_transitions)
```

[62]:
```
plot_gaussians(trained_states)
```


10
5
0
−5
−10
−15
−20
−10
0
10
20
a
i
l
n