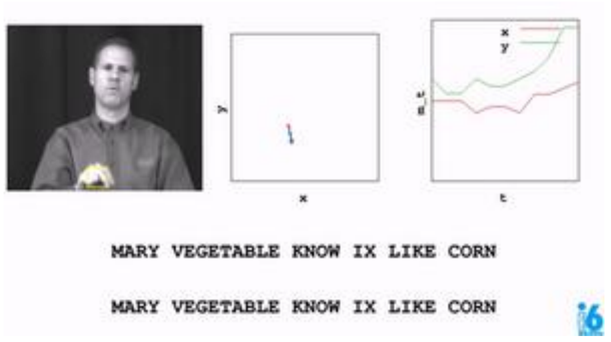
Artificial Intelligence Engineer Nanodegree - Probabilistic Models

Project: Sign Language Recognition System

Introduction

The overall goal of this project is to build a word recognizer for American Sign Language video sequences, demonstrating the power of probabalistic models. In particular, this project employs hidden Markov models (HMM's) to analyze a series of measurements taken from videos of American Sign Language (ASL) collected for research (see the RWTH-BOSTON-104 Database). In this video, the right-hand x and y


y
x
t
MARY VEGETABLE KNOW IX LIKE CORN
MARY VEGETABLE KNOW IX LIKE CORN

locations are plotted as the speaker signs the sentence.

The raw data, train, and test sets are pre-defined. You will derive a variety of feature sets (explored in Part 1), as well as implement three different model selection criterion to determine the optimal number of hidden states for each word model (explored in Part 2). Finally, in Part 3 you will implement the recognizer and compare the effects the different combinations of feature sets and model selection criteria.

At the end of each Part, complete the submission cells with implementations, answer all questions, and pass the unit tests. Then submit the completed notebook for review!

PART 1: Data

Features Tutorial

Load the initial database

A data handler designed for this database is provided in the student codebase as the `AslDb` class in the `asl_data` module. This handler creates the initial pandas dataframe from the corpus of data included in the `data` directory as well as dictionaries suitable for extracting data in a format friendly to the hmmlearn library. We'll use those to create models in Part 2.

To start, let's set up the initial database and select an example set of features for the training set. At the end of Part 1, you will create additional feature sets for experimentation.

In [1]:

```
import numpy as np
import pandas as pd
from asl_data import AslDb

asl = AslDb() # initializes the database
asl.df.head() # displays the first five rows of the asl database, indexed by
video and frame
```

Out[1]:

		left-x	left-y	right-x	right-y	nose-x	nose-y	speaker
video	**frame**							
98	**0**	149	181	170	175	161	62	woman-1
	1	149	181	170	175	161	62	woman-1
	2	149	181	170	175	161	62	woman-1
	3	149	181	170	175	161	62	woman-1
	4	149	181	170	175	161	62	woman-1

In [2]:

```
asl.df.ix[98,1]  # look at the data available for an individual frame
```

Out[2]:

```
left-x          149
left-y          181
right-x         170
right-y         175
nose-x          161
nose-y           62
speaker     woman-1
Name: (98, 1), dtype: object
```

The frame represented by video 98, frame 1 is shown here:

Feature selection for training the model

The objective of feature selection when training a model is to choose the most relevant variables while keeping the model as simple as possible, thus reducing training time. We can use the raw features already provided or derive our own and add columns to the pandas dataframe `asl.df` for selection. As an example, in the next cell a feature named `'grnd-ry'` is added. This feature is the difference between the right-hand y value and the nose y value, which serves as the "ground" right y value.

In [3]:

```
asl.df['grnd-ry'] = asl.df['right-y'] - asl.df['nose-y']
asl.df.head()  # the new feature 'grnd-ry' is now in the frames dictionary
```

Out[3]:

		left-x	left-y	right-x	right-y	nose-x	nose-y	speaker	grnd-ry
video	**frame**								
98	**0**	149	181	170	175	161	62	woman-1	113
	1	149	181	170	175	161	62	woman-1	113
	2	149	181	170	175	161	62	woman-1	113
	3	149	181	170	175	161	62	woman-1	113
	4	149	181	170	175	161	62	woman-1	113

Try it!

In [4]:

```
from asl_utils import test_features_tryit
# TODO add df columns for 'grnd-rx', 'grnd-ly', 'grnd-lx' representing
differences between hand and nose locations
asl.df['grnd-rx'] = asl.df['right-x'] - asl.df['nose-x']
asl.df['grnd-ly'] = asl.df['left-y'] - asl.df['nose-y']
asl.df['grnd-lx'] = asl.df['left-x'] - asl.df['nose-x']
asl.df.head()
# test the code
test_features_tryit(asl)
```

```
asl.df sample
```

		left-x	left-y	right-x	right-y	nose-x	nose-y	speaker	grnd-ry	grnd-rx	grnd-ly	grnd-lx
video	**frame**											
98	**0**	149	181	170	175	161	62	woman-1	113	9	119	-12
	1	149	181	170	175	161	62	woman-1	113	9	119	-12
	2	149	181	170	175	161	62	woman-1	113	9	119	-12
	3	149	181	170	175	161	62	woman-1	113	9	119	-12
	4	149	181	170	175	161	62	woman-1	113	9	119	-12

Out[4]:

```
Correct!
```

In [5]:

```
# collect the features into a list
features_ground = ['grnd-rx','grnd-ry','grnd-lx','grnd-ly']
 #show a single set of features for a given (video, frame) tuple
[asl.df.ix[98,1][v] for v in features_ground]
```

Out[5]:

```
[9, 113, -12, 119]
```

Build the training set

Now that we have a feature list defined, we can pass that list to the `build_training` method to collect the features for all the words in the training set. Each word in the training set has multiple examples from various videos. Below we can see the unique words that have been loaded into the training set:

In [6]:

```
training = asl.build_training(features_ground)
print("Training words: {}".format(training.words))
```

```
Training words: ['JOHN', 'WRITE', 'HOMEWORK', 'IX-1P', 'SEE', 'YESTERDAY', 'IX', 'LOVE', 'MARY',
'CAN', 'GO', 'GO1', 'FUTURE', 'GO2', 'PARTY', 'FUTURE1', 'HIT', 'BLAME', 'FRED', 'FISH', 'WONT',
'EAT', 'BUT', 'CHICKEN', 'VEGETABLE', 'CHINA', 'PEOPLE', 'PREFER', 'BROCCOLI', 'LIKE', 'LEAVE',
'SAY', 'BUY', 'HOUSE', 'KNOW', 'CORN', 'CORN1', 'THINK', 'NOT', 'PAST', 'LIVE', 'CHICAGO', 'CAR',
'SHOULD', 'DECIDE', 'VISIT', 'MOVIE', 'WANT', 'SELL', 'TOMORROW', 'NEXT-WEEK', 'NEW-YORK', 'LAST-
WEEK', 'WILL', 'FINISH', 'ANN', 'READ', 'BOOK', 'CHOCOLATE', 'FIND', 'SOMETHING-ONE', 'POSS',
'BROTHER', 'ARRIVE', 'HERE', 'GIVE', 'MAN', 'NEW', 'COAT', 'WOMAN', 'GIVE1', 'HAVE', 'FRANK',
'BREAK-DOWN', 'SEARCH-FOR', 'WHO', 'WHAT', 'LEG', 'FRIEND', 'CANDY', 'BLUE', 'SUE', 'BUY1',
'STOLEN', 'OLD', 'STUDENT', 'VIDEOTAPE', 'BORROW', 'MOTHER', 'POTATO', 'TELL', 'BILL', 'THROW',
'APPLE', 'NAME', 'SHOOT', 'SAY-1P', 'SELF', 'GROUP', 'JANA', 'TOY1', 'MANY', 'TOY', 'ALL', 'BOY',
'TEACHER', 'GIRL', 'BOX', 'GIVE2', 'GIVE3', 'GET', 'PUTASIDE']
```

The training data in `training` is an object of class `WordsData` defined in the `asl_data` module. in addition to the `words` list, data can be accessed with the `get_all_sequences`, `get_all_Xlengths`, `get_word_sequences`, and `get_word_Xlengths`methods. We need the `get_word_Xlengths` method to train multiple sequences with the `hmmlearn` library. In the following example, notice that there are two lists; the first is a concatenation of all the sequences(the X portion) and the second is a list of the sequence lengths(the Lengths portion).

More feature sets

So far we have a simple feature set that is enough to get started modeling. However, we might get better results if we manipulate the raw values a bit more, so we will go ahead and set up some other options now for experimentation later. For example, we could normalize each speaker's range of motion with grouped statistics using Pandas stats functions and pandas groupby. Below is an example for finding the means of all speaker subgroups.

In [8]:

```
df_means = asl.df.groupby('speaker').mean()
df_means
```

Out[8]:

	left-x	left-y	right-x	right-y	nose-x	nose-y	grnd-ry	grnd-rx	grnd-ly	g
speaker										
man-1	206.248203	218.679449	155.464350	150.371031	175.031756	61.642600	88.728430	-19.567406	157.036848	3
woman-1	164.661438	161.271242	151.017865	117.332462	162.655120	57.245098	60.087364	-11.637255	104.026144	2
woman-2	183.214509	176.527232	156.866295	119.835714	170.318973	58.022098	61.813616	-13.452679	118.505134	1

To select a mean that matches by speaker, use the pandas map method:

In [9]:

```
asl.df['left-x-mean']= asl.df['speaker'].map(df_means['left-x'])
asl.df['left-y-mean']= asl.df['speaker'].map(df_means['left-y'])
asl.df['right-x-mean']= asl.df['speaker'].map(df_means['right-x'])
asl.df['right-y-mean']= asl.df['speaker'].map(df_means['right-y'])
asl.df['nose-x-mean']= asl.df['speaker'].map(df_means['nose-x'])
asl.df['nose-y-mean']= asl.df['speaker'].map(df_means['nose-y'])
asl.df['grnd-ry-mean']= asl.df['speaker'].map(df_means['grnd-ry'])
asl.df['grnd-rx-mean']= asl.df['speaker'].map(df_means['grnd-rx'])
asl.df['grnd-ly-mean']= asl.df['speaker'].map(df_means['grnd-ly'])
asl.df['grnd-lx-mean']= asl.df['speaker'].map(df_means['grnd-lx'])
asl.df.head()
```

Out[9]:

		left-x	left-y	right-x	right-y	nose-x	nose-y	speaker	grnd-ry	grnd-rx	grnd-ly	...	left-x-mean	left-y-mean	right-x-mean
video	frame														
98	0	149	181	170	175	161	62	woman-1	113	9	119	...	164.661438	161.271242	151.0178
	1	149	181	170	175	161	62	woman-1	113	9	119	...	164.661438	161.271242	151.0178
	2	149	181	170	175	161	62	woman-1	113	9	119	...	164.661438	161.271242	151.0178
	3	149	181	170	175	161	62	woman-1	113	9	119	...	164.661438	161.271242	151.0178
	4	149	181	170	175	161	62	woman-1	113	9	119	...	164.661438	161.271242	151.0178

5 rows × 21 columns

Try it!

In [10]:

```
from asl_utils import test_std_tryit
# TODO Create a dataframe named `df_std` with standard deviations grouped by
speaker
df_std = asl.df.groupby('speaker').std()
asl.df['left-x-std']= asl.df['speaker'].map(df_std['left-x'])
asl.df['left-y-std']= asl.df['speaker'].map(df_std['left-y'])
asl.df['right-x-std']= asl.df['speaker'].map(df_std['right-x'])
asl.df['right-y-std']= asl.df['speaker'].map(df_std['right-y'])
asl.df['nose-x-std']= asl.df['speaker'].map(df_std['nose-x'])
asl.df['nose-y-std']= asl.df['speaker'].map(df_std['nose-y'])
asl.df['grnd-ry-std']= asl.df['speaker'].map(df_std['grnd-ry'])
asl.df['grnd-rx-std']= asl.df['speaker'].map(df_std['grnd-rx'])
asl.df['grnd-ly-std']= asl.df['speaker'].map(df_std['grnd-ly'])
asl.df['grnd-lx-std']= asl.df['speaker'].map(df_std['grnd-lx'])
df_std
# test the code
test_std_tryit(df_std)
```

df_std

	left-x	left-y	right-x	right-y	nose-x	nose-y	grnd-ry	grnd-rx	grnd-ly	grnd-lx
speaker										
man-1	15.154425	36.328485	18.901917	54.902340	6.654573	5.520045	53.487999	20.269032	36.572749	15.080360
woman-1	17.573442	26.594521	16.459943	34.667787	3.549392	3.538330	33.972660	16.764706	27.117393	17.328941
woman-2	15.388711	28.825025	14.890288	39.649111	4.099760	3.416167	39.128572	16.191324	29.320655	15.050938

Out[10]: Correct!

Features Implementation Submission

HMM SLG

Implement four feature sets and answer the question that follows.

- normalized Cartesian coordinates
  - use *mean* and *standard deviation* statistics and the standard score equation to account for speakers with different heights and arm length
- polar coordinates
  - calculate polar coordinates with Cartesian to polar equations
  - use the np.arctan2 function and *swap the x and y axes* to move the 00 to 2π2π discontinuity to 12 o'clock instead of 3 o'clock; in other words, the normal break in radians value from 00 to 2π2π occurs directly to the left of the speaker's nose, which may be in the signing area and interfere with results. By swapping the x and y axes, that discontinuity move to directly above the speaker's head, an area not generally used in signing.
- delta difference
  - as described in Thad's lecture, use the difference in values between one frame and the next frames as features
  - pandas diff method and fillna method will be helpful for this one
- custom features
  - These are your own design; combine techniques used above or come up with something else entirely. We look forward to seeing what you come up with! Some ideas to get you started:
    - normalize using a feature scaling equation
    - normalize the polar coordinates
    - adding additional deltas

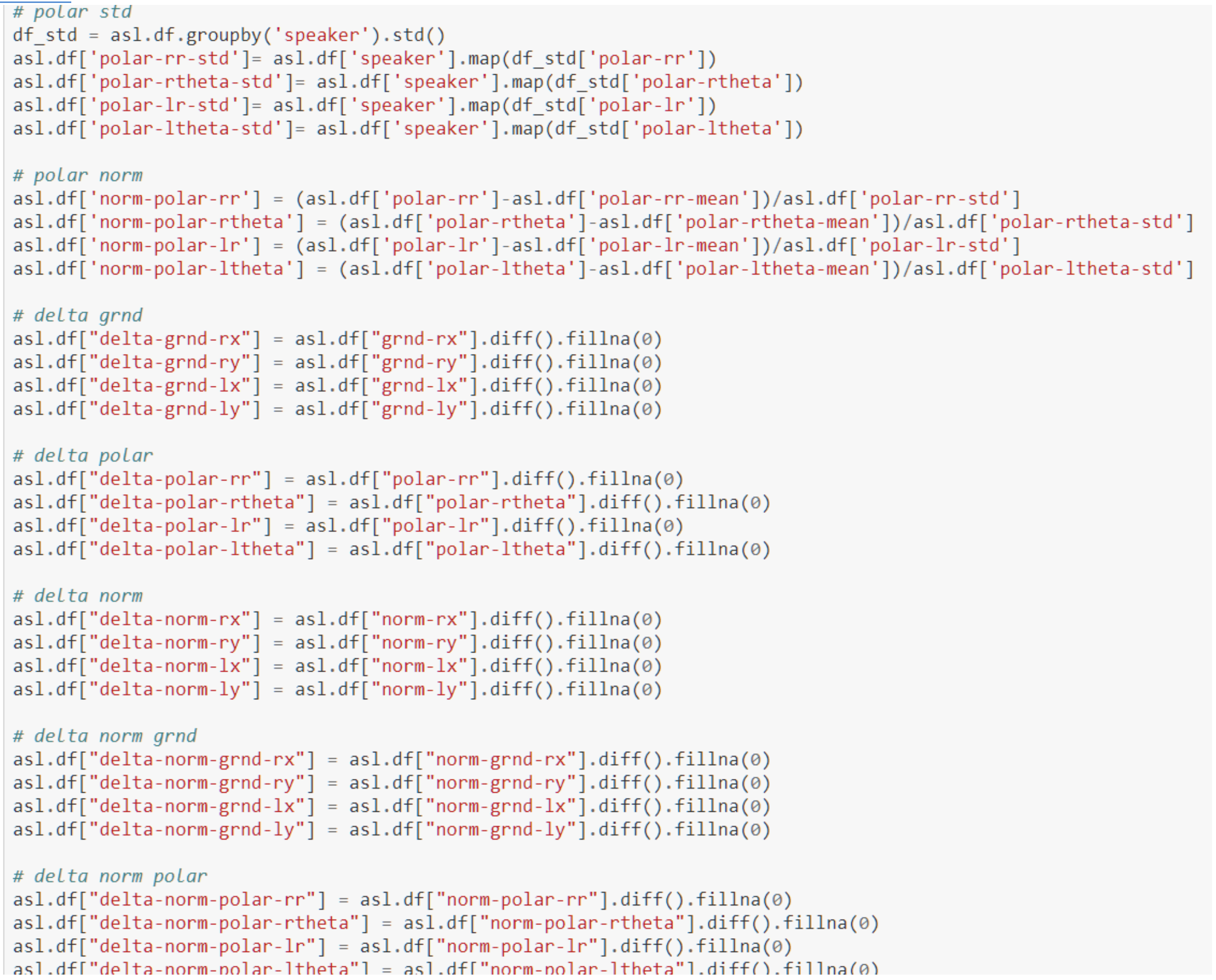
In [11]:
```
# TODO add features for normalized by speaker values of left, right, x, y
# Name these 'norm-rx', 'norm-ry', 'norm-lx', and 'norm-ly'
# using Z-score scaling (X-Xmean)/Xstd
asl.df['norm-rx'] = (asl.df['right-x']-asl.df['right-x-mean'])/asl.df['right-x-std']
asl.df['norm-ry'] = (asl.df['right-y']-asl.df['right-y-mean'])/asl.df['right-y-std']
asl.df['norm-lx'] = (asl.df['left-x']-asl.df['left-x-mean'])/asl.df['left-x-std']
asl.df['norm-ly'] = (asl.df['left-y']-asl.df['left-y-mean'])/asl.df['left-y-std']

features_norm = ['norm-rx', 'norm-ry', 'norm-lx','norm-ly']
```

In [12]:
```
# TODO add features for polar coordinate values where the nose is the origin
# Name these 'polar-rr', 'polar-rtheta', 'polar-lr', and 'polar-ltheta'
# Note that 'polar-rr' and 'polar-rtheta' refer to the radius and angle
asl.df['polar-rr'] = np.sqrt(asl.df['grnd-rx']**2 + asl.df['grnd-ry']**2)
asl.df['polar-rtheta'] = np.arctan2(asl.df['grnd-rx'], asl.df['grnd-ry'])
asl.df['polar-lr'] = np.sqrt(asl.df['grnd-lx']**2 + asl.df['grnd-ly']**2)
asl.df['polar-ltheta'] = np.arctan2(asl.df['grnd-lx'], asl.df['grnd-ly'])

features_polar = ['polar-rr', 'polar-rtheta', 'polar-lr', 'polar-ltheta']
```

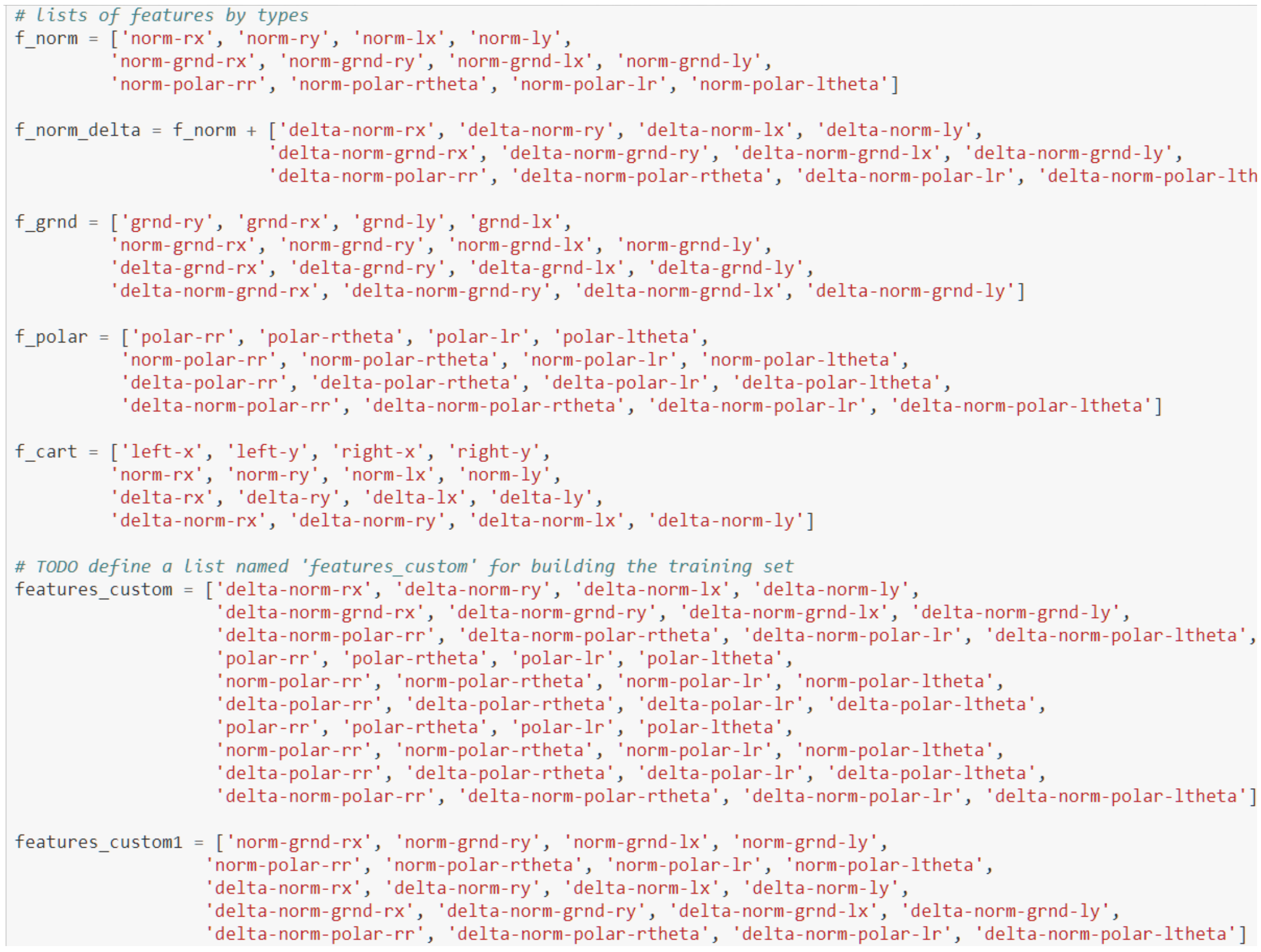
In [13]:
```
# TODO add features for left, right, x, y differences by one time step,
# i.e. the "delta" values discussed in the lecture
# Name these 'delta-rx', 'delta-ry', 'delta-lx', and 'delta-ly'
asl.df["delta-rx"] = asl.df["right-x"].diff().fillna(0)
asl.df["delta-ry"] = asl.df["right-y"].diff().fillna(0)
asl.df["delta-lx"] = asl.df["left-x"].diff().fillna(0)
asl.df["delta-ly"] = asl.df["left-y"].diff().fillna(0)

features_delta = ['delta-rx', 'delta-ry', 'delta-lx', 'delta-ly']
```

In [14]:
```
# TODO add features of your own design, which may be a combination of the above or something else
# Name these whatever you would like

# grnd norm
asl.df['norm-grnd-rx'] = (asl.df['grnd-rx']-asl.df['grnd-rx-mean'])/asl.df['grnd-rx-std']
asl.df['norm-grnd-ry'] = (asl.df['grnd-ry']-asl.df['grnd-ry-mean'])/asl.df['grnd-ry-std']
asl.df['norm-grnd-lx'] = (asl.df['grnd-lx']-asl.df['grnd-lx-mean'])/asl.df['grnd-lx-std']
asl.df['norm-grnd-ly'] = (asl.df['grnd-ly']-asl.df['grnd-ly-mean'])/asl.df['grnd-ly-std']

# polar mean
df_means = asl.df.groupby('speaker').mean()
asl.df['polar-rr-mean']= asl.df['speaker'].map(df_means['polar-rr'])
asl.df['polar-rtheta-mean']= asl.df['speaker'].map(df_means['polar-rtheta'])
asl.df['polar-lr-mean']= asl.df['speaker'].map(df_means['polar-lr'])
asl.df['polar-ltheta-mean']= asl.df['speaker'].map(df_means['polar-ltheta'])
```

HMM SLG



```
# polar std
df_std = asl.df.groupby('speaker').std()
asl.df['polar-rr-std']= asl.df['speaker'].map(df_std['polar-rr'])
asl.df['polar-rtheta-std']= asl.df['speaker'].map(df_std['polar-rtheta'])
asl.df['polar-lr-std']= asl.df['speaker'].map(df_std['polar-lr'])
asl.df['polar-ltheta-std']= asl.df['speaker'].map(df_std['polar-ltheta'])

# polar norm
asl.df['norm-polar-rr'] = (asl.df['polar-rr']-asl.df['polar-rr-mean'])/asl.df['polar-rr-std']
asl.df['norm-polar-rtheta'] = (asl.df['polar-rtheta']-asl.df['polar-rtheta-mean'])/asl.df['polar-rtheta-std']
asl.df['norm-polar-lr'] = (asl.df['polar-lr']-asl.df['polar-lr-mean'])/asl.df['polar-lr-std']
asl.df['norm-polar-ltheta'] = (asl.df['polar-ltheta']-asl.df['polar-ltheta-mean'])/asl.df['polar-ltheta-std']

# delta grnd
asl.df["delta-grnd-rx"] = asl.df["grnd-rx"].diff().fillna(0)
asl.df["delta-grnd-ry"] = asl.df["grnd-ry"].diff().fillna(0)
asl.df["delta-grnd-lx"] = asl.df["grnd-lx"].diff().fillna(0)
asl.df["delta-grnd-ly"] = asl.df["grnd-ly"].diff().fillna(0)

# delta polar
asl.df["delta-polar-rr"] = asl.df["polar-rr"].diff().fillna(0)
asl.df["delta-polar-rtheta"] = asl.df["polar-rtheta"].diff().fillna(0)
asl.df["delta-polar-lr"] = asl.df["polar-lr"].diff().fillna(0)
asl.df["delta-polar-ltheta"] = asl.df["polar-ltheta"].diff().fillna(0)

# delta norm
asl.df["delta-norm-rx"] = asl.df["norm-rx"].diff().fillna(0)
asl.df["delta-norm-ry"] = asl.df["norm-ry"].diff().fillna(0)
asl.df["delta-norm-lx"] = asl.df["norm-lx"].diff().fillna(0)
asl.df["delta-norm-ly"] = asl.df["norm-ly"].diff().fillna(0)

# delta norm grnd
asl.df["delta-norm-grnd-rx"] = asl.df["norm-grnd-rx"].diff().fillna(0)
asl.df["delta-norm-grnd-ry"] = asl.df["norm-grnd-ry"].diff().fillna(0)
asl.df["delta-norm-grnd-lx"] = asl.df["norm-grnd-lx"].diff().fillna(0)
asl.df["delta-norm-grnd-ly"] = asl.df["norm-grnd-ly"].diff().fillna(0)

# delta norm polar
asl.df["delta-norm-polar-rr"] = asl.df["norm-polar-rr"].diff().fillna(0)
asl.df["delta-norm-polar-rtheta"] = asl.df["norm-polar-rtheta"].diff().fillna(0)
asl.df["delta-norm-polar-lr"] = asl.df["norm-polar-lr"].diff().fillna(0)
asl.df["delta-norm-polar-ltheta"] = asl.df["norm-polar-ltheta"].diff().fillna(0)
```



```
# lists of features by types
f_norm = ['norm-rx', 'norm-ry', 'norm-lx', 'norm-ly',
          'norm-grnd-rx', 'norm-grnd-ry', 'norm-grnd-lx', 'norm-grnd-ly',
          'norm-polar-rr', 'norm-polar-rtheta', 'norm-polar-lr', 'norm-polar-ltheta']

f_norm_delta = f_norm + ['delta-norm-rx', 'delta-norm-ry', 'delta-norm-lx', 'delta-norm-ly',
                         'delta-norm-grnd-rx', 'delta-norm-grnd-ry', 'delta-norm-grnd-lx', 'delta-norm-grnd-ly',
                         'delta-norm-polar-rr', 'delta-norm-polar-rtheta', 'delta-norm-polar-lr', 'delta-norm-polar-lth

f_grnd = ['grnd-ry', 'grnd-rx', 'grnd-ly', 'grnd-lx',
          'norm-grnd-rx', 'norm-grnd-ry', 'norm-grnd-lx', 'norm-grnd-ly',
          'delta-grnd-rx', 'delta-grnd-ry', 'delta-grnd-lx', 'delta-grnd-ly',
          'delta-norm-grnd-rx', 'delta-norm-grnd-ry', 'delta-norm-grnd-lx', 'delta-norm-grnd-ly']

f_polar = ['polar-rr', 'polar-rtheta', 'polar-lr', 'polar-ltheta',
           'norm-polar-rr', 'norm-polar-rtheta', 'norm-polar-lr', 'norm-polar-ltheta',
           'delta-polar-rr', 'delta-polar-rtheta', 'delta-polar-lr', 'delta-polar-ltheta',
           'delta-norm-polar-rr', 'delta-norm-polar-rtheta', 'delta-norm-polar-lr', 'delta-norm-polar-ltheta']

f_cart = ['left-x', 'left-y', 'right-x', 'right-y',
          'norm-rx', 'norm-ry', 'norm-lx', 'norm-ly',
          'delta-rx', 'delta-ry', 'delta-lx', 'delta-ly',
          'delta-norm-rx', 'delta-norm-ry', 'delta-norm-lx', 'delta-norm-ly']

# TODO define a list named 'features_custom' for building the training set
features_custom = ['delta-norm-rx', 'delta-norm-ry', 'delta-norm-lx', 'delta-norm-ly',
                   'delta-norm-grnd-rx', 'delta-norm-grnd-ry', 'delta-norm-grnd-lx', 'delta-norm-grnd-ly',
                   'delta-norm-polar-rr', 'delta-norm-polar-rtheta', 'delta-norm-polar-lr', 'delta-norm-polar-ltheta',
                   'polar-rr', 'polar-rtheta', 'polar-lr', 'polar-ltheta',
                   'norm-polar-rr', 'norm-polar-rtheta', 'norm-polar-lr', 'norm-polar-ltheta',
                   'delta-polar-rr', 'delta-polar-rtheta', 'delta-polar-lr', 'delta-polar-ltheta',
                   'polar-rr', 'polar-rtheta', 'polar-lr', 'polar-ltheta',
                   'norm-polar-rr', 'norm-polar-rtheta', 'norm-polar-lr', 'norm-polar-ltheta',
                   'delta-polar-rr', 'delta-polar-rtheta', 'delta-polar-lr', 'delta-polar-ltheta',
                   'delta-norm-polar-rr', 'delta-norm-polar-rtheta', 'delta-norm-polar-lr', 'delta-norm-polar-ltheta']

features_custom1 = ['norm-grnd-rx', 'norm-grnd-ry', 'norm-grnd-lx', 'norm-grnd-ly',
                  'norm-polar-rr', 'norm-polar-rtheta', 'norm-polar-lr', 'norm-polar-ltheta',
                  'delta-norm-rx', 'delta-norm-ry', 'delta-norm-lx', 'delta-norm-ly',
                  'delta-norm-grnd-rx', 'delta-norm-grnd-ry', 'delta-norm-grnd-lx', 'delta-norm-grnd-ly',
                  'delta-norm-polar-rr', 'delta-norm-polar-rtheta', 'delta-norm-polar-lr', 'delta-norm-polar-ltheta']
```

```
full_list_features = asl.df.columns[7:]
full_list_features
```

Out[15]:

```
Index(['grnd-ry', 'grnd-rx', 'grnd-ly', 'grnd-lx', 'left-x-mean',
       'left-y-mean', 'right-x-mean', 'right-y-mean', 'nose-x-mean',
       'nose-y-mean', 'grnd-ry-mean', 'grnd-rx-mean', 'grnd-ly-mean',
       'grnd-lx-mean', 'left-x-std', 'left-y-std', 'right-x-std',
       'right-y-std', 'nose-x-std', 'nose-y-std', 'grnd-ry-std', 'grnd-rx-std',
       'grnd-ly-std', 'grnd-lx-std', 'norm-rx', 'norm-ry', 'norm-lx',
       'norm-ly', 'polar-rr', 'polar-rtheta', 'polar-lr', 'polar-ltheta',
       'delta-rx', 'delta-ry', 'delta-lx', 'delta-ly', 'norm-grnd-rx',
       'norm-grnd-ry', 'norm-grnd-lx', 'norm-grnd-ly', 'polar-rr-mean',
       'polar-rtheta-mean', 'polar-lr-mean', 'polar-ltheta-mean',
       'polar-rr-std', 'polar-rtheta-std', 'polar-lr-std', 'polar-ltheta-std',
       'norm-polar-rr', 'norm-polar-rtheta', 'norm-polar-lr',
       'norm-polar-ltheta', 'delta-grnd-rx', 'delta-grnd-ry', 'delta-grnd-lx',
       'delta-grnd-ly', 'delta-polar-rr', 'delta-polar-rtheta',
       'delta-polar-lr', 'delta-polar-ltheta', 'delta-norm-rx',
       'delta-norm-ry', 'delta-norm-lx', 'delta-norm-ly', 'delta-norm-grnd-rx',
       'delta-norm-grnd-ry', 'delta-norm-grnd-lx', 'delta-norm-grnd-ly',
       'delta-norm-polar-rr', 'delta-norm-polar-rtheta', 'delta-norm-polar-lr',
       'delta-norm-polar-ltheta'],
      dtype='object')
```

Question 1: What custom features did you choose for the features_custom set and why?

Answer 1:

The polar coordinate system is especially useful in cases where the relationship between points is easier to depict in the form of radii and angles. I aimed to highlight the variation in terms of the relationship between the mean and the deviation from the mean.

The normalizing reduces the noise and variance.

The purpose of the delta is to take advantage of the time dimensional information and detect acceleration and deceleration.

All features provide different measurements of the movement which will be interesting to compare the combinations of features and model selectors later to choose the best.

Features Unit Testing

Run the following unit tests as a sanity check on the defined "ground", "norm", "polar", and 'delta" feature sets. The test simply looks for some valid values but is not exhaustive. However, the project should not be submitted if these tests don't pass.

In [16]:

```
import unittest
# import numpy as np

class TestFeatures(unittest.TestCase):

    def test_features_ground(self):
        sample = (asl.df.ix[98, 1][features_ground]).tolist()
        self.assertEqual(sample, [9, 113, -12, 119])
    def test_features_norm(self):
        sample = (asl.df.ix[98, 1][features_norm]).tolist()
        np.testing.assert_almost_equal(sample, [ 1.153,  1.663, -0.891,
0.742], 3)

    def test_features_polar(self):
        sample = (asl.df.ix[98,1][features_polar]).tolist()
        np.testing.assert_almost_equal(sample, [113.3578, 0.0794, 119.603, -
0.1005], 3)

    def test_features_delta(self):
        sample = (asl.df.ix[98, 0][features_delta]).tolist()
        self.assertEqual(sample, [0, 0, 0, 0])
        sample = (asl.df.ix[98, 18][features_delta]).tolist()
        self.assertTrue(sample in [[-16, -5, -2, 4], [-14, -9, 0, 0]], "Sample
value found was {}".format(sample))

suite = unittest.TestLoader().loadTestsFromModule(TestFeatures())
unittest.TextTestRunner().run(suite)
```

```
....
----------------------------------------------------------------------
Ran 4 tests in 0.019s

OK
```

Out[16]:

```
<unittest.runner.TextTestResult run=4 errors=0 failures=0>
```

PART 2: Model Selection

Model Selection Tutorial

The objective of Model Selection is to tune the number of states for each word HMM prior to testing on unseen data. In this section you will explore three methods:

- Log likelihood using cross-validation folds (CV)
- Bayesian Information Criterion (BIC)
- Discriminative Information Criterion (DIC)

Train a single word

Now that we have built a training set with sequence data, we can "train" models for each word. As a simple starting example, we train a single word using Gaussian hidden Markov models (HMM). By using the `fit` method during training, the Baum-Welch Expectation-Maximization (EM) algorithm is invoked iteratively to find the best estimate for the model *for the number of hidden states specified* from a group of sample seequences. For this example, we *assume* the correct number of hidden states is 3, but that is just a guess. How do we know what the "best" number of states for training is? We will need to find some model selection technique to choose the best parameter.

In [17]:

```
import warnings
from hmmlearn.hmm import GaussianHMM

def train_a_word(word, num_hidden_states, features):

    warnings.filterwarnings("ignore", category=DeprecationWarning)
    training = asl.build_training(features)
    X, lengths = training.get_word_Xlengths(word)
    model = GaussianHMM(n_components=num_hidden_states, n_iter=1000).fit(X,
lengths)
    logL = model.score(X, lengths)
    return model, logL

demoword = 'BOOK'
model, logL = train_a_word(demoword, 3, features_ground)
print("Number of states trained in model for {} is {}".format(demoword,
model.n_components))
print("logL = {}".format(logL))
```

```
Number of states trained in model for BOOK is 3
logL = -2331.1138127433223
```

The HMM model has been trained and information can be pulled from the model, including means and variances for each feature and hidden state. The log likelihood for any individual sample or group of samples can also be calculated with the `score` method.

In [18]:

```
def show_model_stats(word, model):
    print("Number of states trained in model for {} is {}".format(word,
model.n_components))
    variance=np.array([np.diag(model.covars_[i]) for i in
range(model.n_components)])
    for i in range(model.n_components):  # for each hidden state
        print("hidden state #{}".format(i))
        print("mean = ", model.means_[i])
        print("variance = ", variance[i])
        print()

show_model_stats(demoword, model)
```

```
Number of states trained in model for BOOK is 3
hidden state #0
mean =  [ -3.46504869  50.66686933  14.02391587  52.04731066]
variance =  [ 49.12346305  43.04799144  39.35109609  47.24195772]

hidden state #1
mean =  [ -11.45300909   94.109178     19.03512475  102.2030162 ]
variance =  [  77.403668    203.35441965   26.68898447  156.12444034]

hidden state #2
mean =  [ -1.12415027  69.44164191  17.02866283  77.7231196 ]
variance =  [ 19.70434594  16.83041492  30.51552305  11.03678246]
```

Try it! Experiment by changing the feature set, word, and/or num_hidden_states values in the next cell to see changes in values.

In [19]:

```
my_testword = 'CHOCOLATE'
model, logL = train_a_word(my_testword, 3, features_ground) # Experiment here
with different parameters
show_model_stats(my_testword, model)
print("logL = {}".format(logL))
```

```
Number of states trained in model for CHOCOLATE is 3
hidden state #0
mean =  [ -9.30211403  55.32333876   6.92259936  71.24057775]
variance =  [ 16.16920957  46.50917372   3.81388185  15.79446427]

hidden state #1
mean =  [ -5.40587658  60.1652424    2.32479599  91.3095432 ]
variance =  [   7.95073876   64.13103127   13.68077479  129.5912395 ]

hidden state #2
mean =  [   0.58333333   87.91666667   12.75         108.5        ]
variance =  [  39.41055556   18.74388889    9.855       144.4175    ]

logL = -601.3291470028639
```

Visualize the hidden states

We can plot the means and variances for each state and feature. Try varying the number of states trained for the HMM model and examine the variances. Are there some models that are "better" than others? How can you tell? We would like to hear what you think in the classroom online.

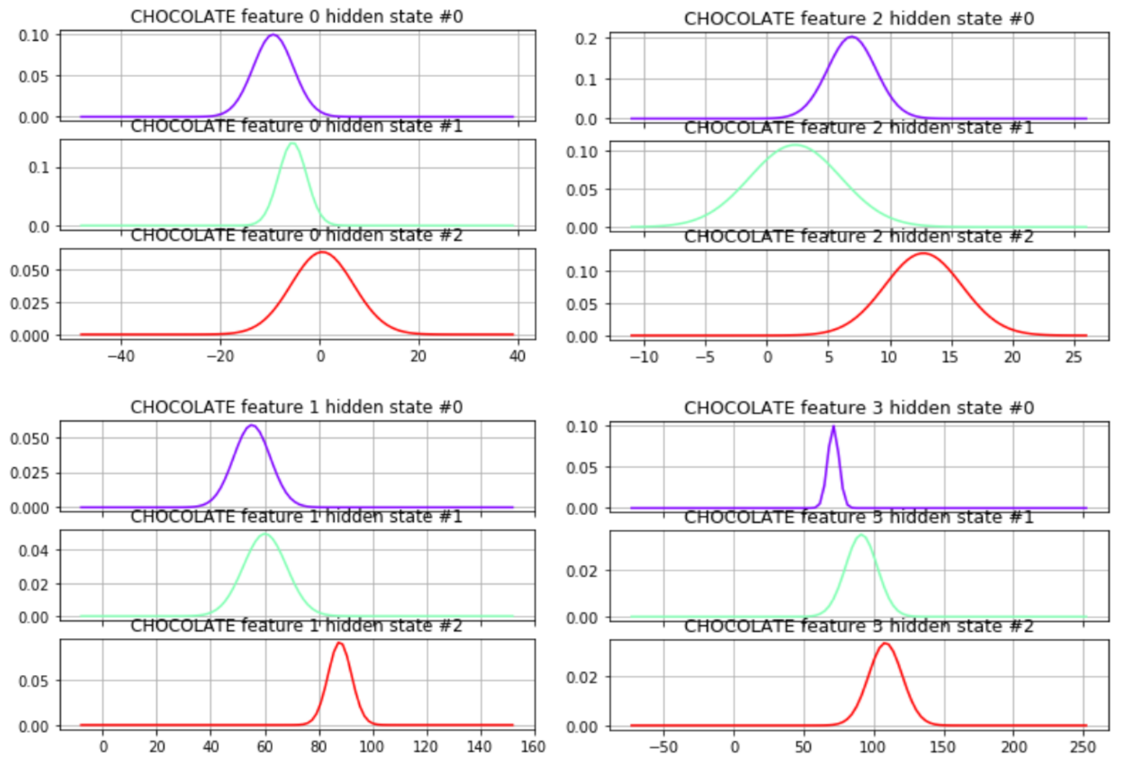
In [20]:

```
%matplotlib inline
```

In [21]:

```
import math
from matplotlib import (cm, pyplot as plt, mlab)

def visualize(word, model):
    """ visualize the input model for a particular word """
    variance=np.array([np.diag(model.covars_[i]) for i in
range(model.n_components)])
    figures = []
    for parm_idx in range(len(model.means_[0])):
        xmin = int(min(model.means_[:,parm_idx]) - max(variance[:,parm_idx]))
        xmax = int(max(model.means_[:,parm_idx]) + max(variance[:,parm_idx]))
        fig, axs = plt.subplots(model.n_components, sharex=True, sharey=False)
        colours = cm.rainbow(np.linspace(0, 1, model.n_components))
        for i, (ax, colour) in enumerate(zip(axs, colours)):
            x = np.linspace(xmin, xmax, 100)
            mu = model.means_[i,parm_idx]
            sigma = math.sqrt(np.diag(model.covars_[i])[parm_idx])
            ax.plot(x, mlab.normpdf(x, mu, sigma), c=colour)
            ax.set_title("{} feature {} hidden state #{}".format(word,
parm_idx, i))
```

```
            ax.grid(True)
        figures.append(plt)
    for p in figures:
        p.show()

visualize(my_testword, model)
```



ModelSelector class

Review the `ModelSelector` class from the codebase found in the `my_model_selectors.py` module. It is designed to be a strategy pattern for choosing different model selectors. For the project submission in this section, subclass `SelectorModel` to implement the following model selectors. In other words, you will write your own classes/functions in the `my_model_selectors.py` module and run them from this notebook:

- `SelectorCV`: Log likelihood with CV
- `SelectorBIC`: BIC
- `SelectorDIC`: DIC

You will train each word in the training set with a range of values for the number of hidden states, and then score these alternatives with the model selector, choosing the "best" according to each strategy. The simple case of training with a constant value for `n_components` can be called using the provided `SelectorConstant` subclass as follow:

In [22]:

```
from my_model_selectors import SelectorConstant

training = asl.build_training(f_cart)  # Experiment here with different feature
sets defined in part 1
word = 'VIDEOTAPE' # Experiment here with different words
model = SelectorConstant(training.get_all_sequences(),
training.get_all_Xlengths(), word, n_constant=3).select()
print("Number of states trained in model for {} is {}".format(word,
model.n_components))
```

```
Number of states trained in model for VIDEOTAPE is 3
```

Cross-validation folds

If we simply score the model with the Log Likelihood calculated from the feature sequences it has been trained on, we should expect that more complex models will have higher likelihoods. However, that doesn't tell us which would have a better likelihood score on unseen data. The model will likely be overfit as complexity is added. To estimate which topology model is better using only the training data, we can compare scores using cross-validation. One technique for cross-validation is to break the training set into "folds" and rotate which fold is left out of training. The "left out" fold scored. This gives us a proxy method of finding the best model to use on "unseen data". In the following example, a set of word sequences is broken into three folds using the scikit-learn Kfold class object. When you implement `SelectorCV`, you will use this technique.

In [23]:

```
from sklearn.model_selection import KFold

training = asl.build_training(f_norm) # Experiment here with different feature
sets
word = 'BOOK' # Experiment here with different words
word_sequences = training.get_word_sequences(word)
split_method = KFold()
for cv_train_idx, cv_test_idx in split_method.split(word_sequences):
    print("Train fold indices:{} Test fold indices:{}".format(cv_train_idx,
cv_test_idx))  # view indices of the folds
```

```
Train fold indices:[ 6  7  8  9 10 11 12 13 14 15 16 17] Test fold indices:[0 1
2 3 4 5]
Train fold indices:[ 0  1  2  3  4  5 12 13 14 15 16 17] Test fold indices:[ 6
7  8  9 10 11]
Train fold indices:[ 0  1  2  3  4  5  6  7  8  9 10 11] Test fold indices:[12
13 14 15 16 17]
```

Tip: In order to run `hmmlearn` training using the X,lengths tuples on the new folds, subsets must be combined based on the indices given for the folds. A helper utility has been provided in the `asl_utils` module named `combine_sequences` for this purpose.

Scoring models with other criterion

Scoring model topologies with **BIC** balances fit and complexity within the training set for each word. In the BIC equation, a penalty term penalizes complexity to avoid overfitting, so that it is not necessary to also use cross-validation in the selection process. There are a number of references on the internet for this criterion. These slides include a formula you may find helpful for your implementation.

The advantages of scoring model topologies with **DIC** over BIC are presented by Alain Biem in this reference (also found here). DIC scores the discriminant ability of a training set for one word against competing words. Instead of a penalty term for complexity, it provides a penalty if model liklihoods for non-matching words are too similar to model likelihoods for the correct word in the word set.

Model Selection Implementation Submission

Implement `SelectorCV`, `SelectorBIC`, and `SelectorDIC` classes in the `my_model_selectors.py` module. Run the selectors on the following five words. Then answer the questions about your results.

Tip: The `hmmlearn` library may not be able to train or score all models. Implement try/except contructs as necessary to eliminate non-viable models from consideration.

In [24]:
```
words_to_train = ['FISH', 'BOOK', 'VEGETABLE', 'FUTURE', 'JOHN']
import timeit
```
In [25]:
```
# TODO: Implement SelectorCV in my_model_selector.py
from my_model_selectors import SelectorCV

training = asl.build_training(features_ground)  # Experiment here with
different feature sets defined in part 1
sequences = training.get_all_sequences()
Xlengths = training.get_all_Xlengths()
for word in words_to_train:
    start = timeit.default_timer()
    model = SelectorCV(sequences, Xlengths, word,
                    min_n_components=2, max_n_components=15, random_state =
14).select()
    end = timeit.default_timer()-start
    if model is not None:
        print("Training complete for {} with {} states with time {}
seconds".format(word, model.n_components, end))
    else:
        print("Training failed for {}".format(word))
```
```
Training complete for FISH with 3 states with time 0.028477233328295014 seconds
Training complete for BOOK with 3 states with time 0.10142323187651267 seconds
Training complete for VEGETABLE with 3 states with time 0.0315744001134555
seconds
Training complete for FUTURE with 3 states with time 0.06779851574847964 seconds
Training complete for JOHN with 3 states with time 0.8805680425010218 seconds
```
In [26]:
```
# TODO: Implement SelectorBIC in module my_model_selectors.py
from my_model_selectors import SelectorBIC

training = asl.build_training(features_ground)  # Experiment here with
different feature sets defined in part 1
sequences = training.get_all_sequences()
Xlengths = training.get_all_Xlengths()
for word in words_to_train:
    start = timeit.default_timer()
    model = SelectorBIC(sequences, Xlengths, word,
                    min_n_components=2, max_n_components=15, random_state =
14).select()
    end = timeit.default_timer()-start
    if model is not None:
        print("Training complete for {} with {} states with time {}
seconds".format(word, model.n_components, end))
    else:
        print("Training failed for {}".format(word))
```
```
Training complete for FISH with 5 states with time 0.4685263662708792 seconds
Training complete for BOOK with 8 states with time 2.6239015997596535 seconds
Training complete for VEGETABLE with 9 states with time 0.9170459401410351
seconds
Training complete for FUTURE with 9 states with time 2.8196575622372357 seconds
Training complete for JOHN with 13 states with time 26.293201254728093 seconds
```

In [27]:

```
# TODO: Implement SelectorDIC in module my_model_selectors.py
from my_model_selectors import SelectorDIC

training = asl.build_training(features_ground)  # Experiment here with
different feature sets defined in part 1
sequences = training.get_all_sequences()
Xlengths = training.get_all_Xlengths()
for word in words_to_train:
    start = timeit.default_timer()
    model = SelectorDIC(sequences, Xlengths, word,
                    min_n_components=2, max_n_components=15, random_state =
14).select()
    end = timeit.default_timer()-start
    if model is not None:
        print("Training complete for {} with {} states with time {}
seconds".format(word, model.n_components, end))
    else:
        print("Training failed for {}".format(word))
```

```
Training complete for FISH with 3 states with time 1.1843782247575518 seconds
Training complete for BOOK with 15 states with time 5.442678855667083 seconds
Training complete for VEGETABLE with 15 states with time 3.9452012808527073
seconds
Training complete for FUTURE with 15 states with time 5.6606024896761085 seconds
Training complete for JOHN with 15 states with time 29.8578638832752 seconds
```

Question 2: Compare and contrast the possible advantages and disadvantages of the various model selectors implemented.

Answer 2:

1. **Cross-validation** is used to evaluate or compare learning algorithms as follows: in each iteration, one or more learning algorithms use k - 1 folds of data to learn one or more models, and subsequently the learned models are asked to make predictions about the data in the validation fold. The performance of each learning algorithm on each fold can be tracked using some predetermined performance metric like accuracy. Upon completion, k samples of the performance metric will be available for each algorithm. Different methodologies such as averaging can be used to obtain an aggregate measure from these sample, or these samples can be used in a statistical hypothesis test to show that one algorithm is superior to another[1]

   pros - Acurate performance; Through usage of data (does not need a lot of data), as the train data is folded to simulate the behavior the model will have in test data

   cons - Small samples of performance estimation; Overlapped training data; Elevated Type I error for comparison; Underestimated performance variance or overestimated degree of freedom for comparison[1]

2. **The Bayesian information criterion (BIC)** or Schwarz criterion (also SBC, SBIC) is a criterion for model selection among a finite set of models; the model with the lowest BIC is preferred. It is based, in part, on the likelihood function and it is closely related to the Akaike information criterion (AIC). When fitting models, it is possible to increase the likelihood by adding parameters, but

doing so may result in overfitting. Both BIC and AIC attempt to resolve this problem by introducing a penalty term for the number of parameters in the model; the penalty term is larger in BIC than in AIC.[2]

- It is independent of the prior or the prior is "vague" (a constant).
- It can measure the efficiency of the parameterized model in terms of predicting the data.
- It penalizes the complexity of the model where complexity refers to the number of parameters in the model.
- It is approximately equal to the minimum description length criterion but with negative sign.
- It can be used to choose the number of clusters according to the intrinsic complexity present in a particular dataset.[2]

The BIC criterion suffers from two main limitations:

- the above approximation is only valid for sample size n much larger than the number k of parameters in the model.
- the BIC cannot handle complex collections of models as in the variable selection (or feature selection) problem in high-dimension.[3]

3. **The deviance information criterion (DIC)** is a hierarchical modeling generalization of the Akaike information criterion (AIC) and the Bayesian information criterion (BIC). It is particularly useful in Bayesian model selection problems where the posterior distributions of the models have been obtained by Markov chain Monte Carlo (MCMC) simulation. Like AIC and BIC, DIC is an asymptotic approximation as the sample size becomes large. It is only valid when the posterior distribution is approximately multivariate normal.[4]

   The idea is that models with smaller DIC should be preferred to models with larger DIC. Models are penalized both by the value of D, which favors a good fit, but also (in common with AIC and BIC) by the effective number of parameters PD,. Since D will decrease as the number of parameters in a model increases, the PD, term compensates for this effect by favoring models with a smaller number of parameters.[4]

   The advantage of DIC over other criteria in the case of Bayesian model selection is that the DIC is easily calculated from the samples generated by a Markov chain Monte Carlo simulation. AIC and BIC require calculating the likelihood at its maximum over **theta**, which is not readily available from the MCMC simulation. But to calculate DIC, simply compute D as the average of D(**theta**), over the samples of **theta**, and D(**theta**) as the value of D, evaluated at the average of the samples of **theta**. Then the DIC follows directly from these approximations.[4]

DIC differs from Bayes factors and BIC in both form and aims. BIC attempts to ientify the ‘true’ model, DIC is not based on any assumption of a ‘true’ model and is concerend with short-term predictive ability. BIC requires specification of the number of parameters, while DIC estimates the effective number of parameters. BIC provides a procedure for model averaging, DIC does not.[5]

I may add that SelectorCV is the only selector that tests unseen data. BIC tries to avoid overfitting by penalizing a large number of features (and states). DIC is useful because evaluates the performance of competing words, it chooses models that get a low score on competing words. This is useful because penalizes the model when it confuses with other words.

[1]PAYAM REFAEILZADEH, LEI TANG, HUAN LIU Arizona State University http://leitang.net/papers/ency-cross-validation.pdf

[2]Wikipedia https://en.wikipedia.org/wiki/Bayesian_information_criterion

[3]Giraud, C. (2015). Introduction to high-dimensional statistics. Chapman & Hall/CRC. ISBN 9781482237948

[4]Wikipedia https://en.wikipedia.org/wiki/Deviance_information_criterion

[5]University of Cambridge https://www.mrc-bsu.cam.ac.uk/software/bugs/the-bugs-project-dic/#q5

Model Selector Unit Testing

Run the following unit tests as a sanity check on the implemented model selectors. The test simply looks for valid interfaces but is not exhaustive. However, the project should not be submitted if these tests don't pass.

In [28]:

```
from asl_test_model_selectors import TestSelectors
suite = unittest.TestLoader().loadTestsFromModule(TestSelectors())
unittest.TextTestRunner().run(suite)
```

```
....
----------------------------------------------------------------------
Ran 4 tests in 24.366s
OK
```

Out[28]:

```
<unittest.runner.TextTestResult run=4 errors=0 failures=0>
```

PART 3: Recognizer

The objective of this section is to "put it all together". Using the four feature sets created and the three model selectors, you will experiment with the models and present your results. Instead of training only five specific words as in the previous section, train the entire set with a feature set and model selector strategy.

Recognizer Tutorial

Train the full training set

The following example trains the entire set with the example `features_ground` and `SelectorConstant` features and model selector. Use this pattern for you experimentation and final submission cells.

In [29]:

```
# autoreload for automatically reloading changes made in my_model_selectors and my_recognizer
%load_ext autoreload
%autoreload 2

from my_model_selectors import SelectorConstant

def train_all_words(features, model_selector):
    training = asl.build_training(features)  # Experiment here with different feature sets defined in part 1
    sequences = training.get_all_sequences()
    Xlengths = training.get_all_Xlengths()
    model_dict = {}
    for word in training.words:
        model = model_selector(sequences, Xlengths, word,
                        n_constant=3).select()
        model_dict[word]=model
    return model_dict
models = train_all_words(features_ground, SelectorConstant)
print("Number of word models returned = {}".format(len(models)))
```

```
Number of word models returned = 112
```

Load the test set

The `build_test` method in `ASLdb` is similar to the `build_training` method already presented, but there are a few differences:

- the object is type `SinglesData`
- the internal dictionary keys are the index of the test word rather than the word itself
- the getter methods are `get_all_sequences`, `get_all_Xlengths`, `get_item_sequences` and `get_item_Xlengths`

In [30]:

```
test_set = asl.build_test(features_ground)
print("Number of test set items: {}".format(test_set.num_items))
print("Number of test set sentences: {}".format(len(test_set.sentences_index)))
```

```
Number of test set items: 178
Number of test set sentences: 40
```

Recognizer Implementation Submission

For the final project submission, students must implement a recognizer following guidance in the `my_recognizer.py` module. Experiment with the four feature sets and the three model selection methods (that's 12 possible combinations). You can add and remove cells for experimentation or run the recognizers locally in some other way during your experiments, but retain the results for your discussion. For submission, you will provide code cells of **only three** interesting combinations for your discussion (see questions below). At least one of these should produce a word error rate of less than 60%, i.e. WER < 0.60 .

Tip: The hmmlearn library may not be able to train or score all models. Implement try/except contructs as necessary to eliminate non-viable models from consideration.

In [31]:

```
# TODO implement the recognize method in my_recognizer
from my_recognizer import recognize
from asl_utils import show_errors
```

In [32]:

```
# TODO Choose a feature set and model selector
features = f_polar # change as needed
model_selector = SelectorCV # change as needed

# TODO Recognize the test set and display the result with the show_errors
method
start = timeit.default_timer()
models = train_all_words(features, model_selector)
test_set = asl.build_test(features)
probabilities, guesses = recognize(models, test_set)
show_errors(guesses, test_set)
end = timeit.default_timer()-start
print('Recognition complete with time {} seconds'.format(end))
```

```
**** WER = 0.4943820224719101
Total correct: 90 out of 178
Video  Recognized                                                    Correct
=====================================================================================================
    2: *POSS WRITE HOMEWORK                                          JOHN WRITE HOMEWORK
    7: JOHN *CAR GO *ARRIVE                                              JOHN CAN GO CAN
..............
```

```
199: *IX CHOCOLATE WHO                                                LIKE
CHOCOLATE WHO
  201: JOHN *MAN *WOMAN *WOMAN BUY HOUSE                                JOHN TELL
MARY IX-1P BUY HOUSE
Recognition complete with time 66.92159488927825 seconds
```

In [49]:

```
# TODO Choose a feature set and model selector
features =  full_list_features # change as needed
model_selector = SelectorDIC # change as needed

# TODO Recognize the test set and display the result with the show_errors
method
start = timeit.default_timer()
models = train_all_words(features, model_selector)
test_set = asl.build_test(features)
probabilities, guesses = recognize(models, test_set)
show_errors(guesses, test_set)
end = timeit.default_timer()-start
print('Recognition complete with time {} seconds'.format(end))
```

```
**** WER = 0.5168539325842697
Total correct: 86 out of 178
Video  Recognized                                                    Correct
================================================================================
    2: JOHN WRITE HOMEWORK                                       JOHN WRITE HOMEWORK
    7: JOHN *CAR GO *WHAT                                        JOHN CAN GO CAN
…….

  199: *JOHN *ARRIVE *JOHN                                       LIKE CHOCOLATE WHO
  201: JOHN *IX *IX *JOHN BUY HOUSE                               JOHN TELL MARY
IX-1P BUY HOUSE
Recognition complete with time 475.11281869701634 seconds
```

In [34]:

```
# TODO Choose a feature set and model selector
features =  features_custom # change as needed
model_selector = SelectorBIC # change as needed

# TODO Recognize the test set and display the result with the show_errors
method
start = timeit.default_timer()
models = train_all_words(features, model_selector)
test_set = asl.build_test(features)
probabilities, guesses = recognize(models, test_set)
show_errors(guesses, test_set)
end = timeit.default_timer()-start
print('Recognition complete with time {} seconds'.format(end))
```

```
**** WER = 0.4157303370786517
Total correct: 104 out of 178
Video  Recognized                                                    Correct
================================================================================
    2: JOHN WRITE HOMEWORK                                       JOHN WRITE HOMEWORK
    7: JOHN *CAR GO CAN                                           JOHN CAN GO CAN
```
…….

```
  199: *JOHN CHOCOLATE WHO                                    LIKE CHOCOLATE WHO
  201: JOHN *WHO MARY *LIKE BUY HOUSE            JOHN TELL MARY IX-1P BUY HOUSE
Recognition complete with time 231.07731067582517 seconds
```

Question 3: Summarize the error results from three combinations of features and model selectors. What was the "best" combination and why? What additional information might we use to improve our WER? For more insight on improving WER, take a look at the introduction to Part 4.

Answer 3:

The main steps of the research:

1.**Prepare lists of features by types:**

f_norm = ['norm-rx', 'norm-ry', 'norm-lx', 'norm-ly', 'norm-grnd-rx', 'norm-grnd-ry', 'norm-grnd-lx', 'norm-grnd-ly', 'norm-polar-rr', 'norm-polar-rtheta', 'norm-polar-lr', 'norm-polar-ltheta']

f_norm_delta = f_norm + ['delta-norm-rx', 'delta-norm-ry', 'delta-norm-lx', 'delta-norm-ly', 'delta-norm-grnd-rx', 'delta-norm-grnd-ry', 'delta-norm-grnd-lx', 'delta-norm-grnd-ly', 'delta-norm-polar-rr', 'delta-norm-polar-rtheta', 'delta-norm-polar-lr', 'delta-norm-polar-ltheta']

f_grnd = ['grnd-ry', 'grnd-rx', 'grnd-ly', 'grnd-lx', 'norm-grnd-rx', 'norm-grnd-ry', 'norm-grnd-lx', 'norm-grnd-ly', 'delta-grnd-rx', 'delta-grnd-ry', 'delta-grnd-lx', 'delta-grnd-ly', 'delta-norm-grnd-rx', 'delta-norm-grnd-ry', 'delta-norm-grnd-lx', 'delta-norm-grnd-ly']

f_polar = ['polar-rr', 'polar-rtheta', 'polar-lr', 'polar-ltheta', 'norm-polar-rr', 'norm-polar-rtheta', 'norm-polar-lr', 'norm-polar-ltheta', 'delta-polar-rr', 'delta-polar-rtheta', 'delta-polar-lr', 'delta-polar-ltheta', 'delta-norm-polar-rr', 'delta-norm-polar-rtheta', 'delta-norm-polar-lr', 'delta-norm-polar-ltheta']

f_cart = ['left-x', 'left-y', 'right-x', 'right-y', 'norm-rx', 'norm-ry', 'norm-lx', 'norm-ly', 'delta-rx', 'delta-ry', 'delta-lx', 'delta-ly', 'delta-norm-rx', 'delta-norm-ry', 'delta-norm-lx', 'delta-norm-ly']

2.**Check the efficiency of each selector to choose most effective combination:**

- by WER

WER	full_list	f_cart	f_grnd	f_polar	f_norm	f_norm_delta
CV	0.7134	0.4831	0.4775	0.4943	0.5898	0.5056
BIC	0.5337	0.4831	0.4607	**0.4326**	0.5955	0.4551
DIC	0.5168	0.4831	0.4606	0.4438	0.5730	0.4887

- by Time

Time	full_list	f_cart	f_grnd	f_polar	f_norm	f_norm_delta
CV	226.3371	62.3689	61.7717	61.2972	50.1182	90.5488
BIC	339.2213	157.616	172.993	**153.4714**	124.531	189.8512
DIC	475.1128	320.437	285.300	324.5010	346.459	361.8028

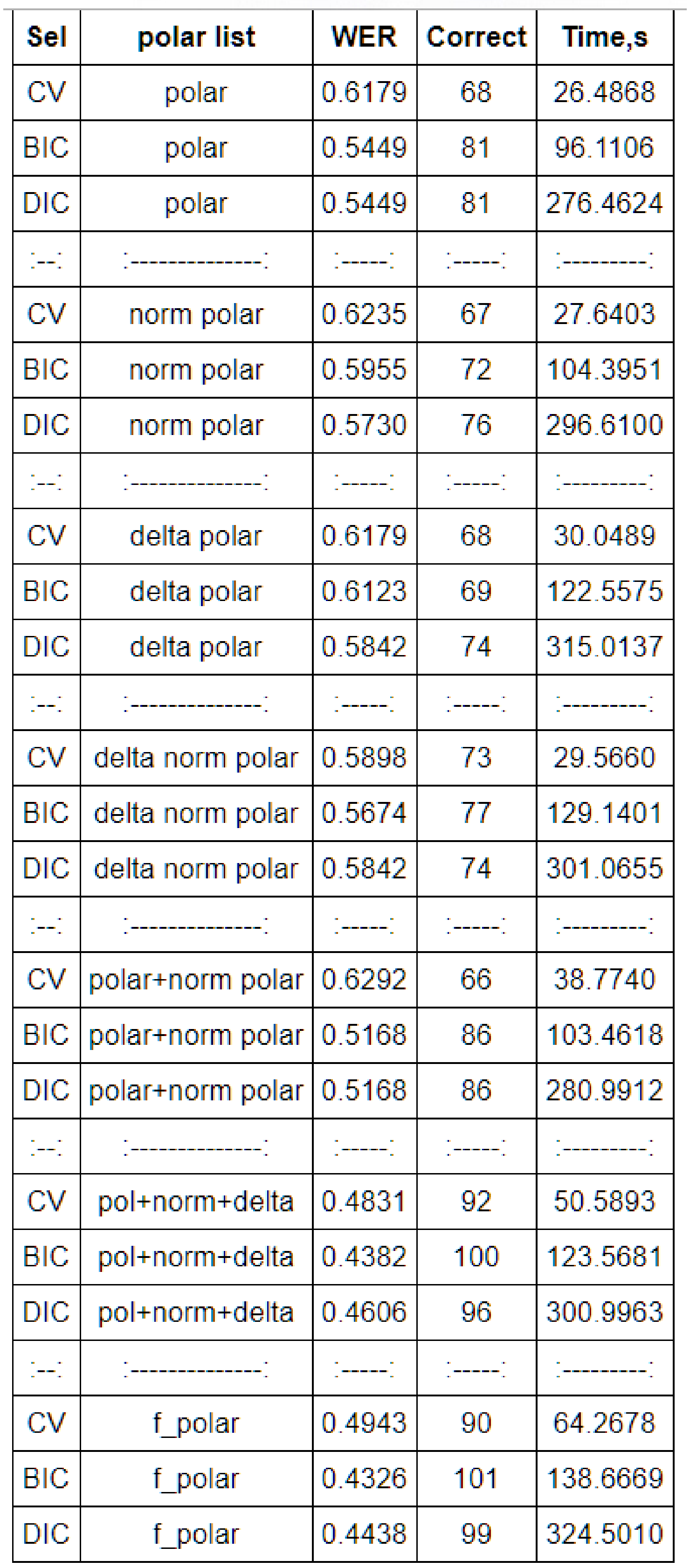
- by Correction

Corr	full_list	f_cart	f_grnd	f_polar	f_norm	f_norm_delta
CV	51	92	93	90	73	88
BIC	83	92	96	**101**	79	97
DIC	86	92	96	99	76	91

as was shown above, the most effective combination is BIC with features for polar coordinate values where the nose is the origin f_polar with WER = 0.4326 and Correction = 101.

In general, the BIC selector seems to be the most effective one with all checked lists of features. So, I'll continue with this selector further.

3.**Research the influence of a new features addition for f_polar list of feautures:**



Sel	polar list	WER	Correct	Time,s
CV	polar	0.6179	68	26.4868
BIC	polar	0.5449	81	96.1106
DIC	polar	0.5449	81	276.4624
:--:	:--------------:	:-----:	:-----:	:---------:
CV	norm polar	0.6235	67	27.6403
BIC	norm polar	0.5955	72	104.3951
DIC	norm polar	0.5730	76	296.6100
:--:	:--------------:	:-----:	:-----:	:---------:
CV	delta polar	0.6179	68	30.0489
BIC	delta polar	0.6123	69	122.5575
DIC	delta polar	0.5842	74	315.0137
:--:	:--------------:	:-----:	:-----:	:---------:
CV	delta norm polar	0.5898	73	29.5660
BIC	delta norm polar	0.5674	77	129.1401
DIC	delta norm polar	0.5842	74	301.0655
:--:	:--------------:	:-----:	:-----:	:---------:
CV	polar+norm polar	0.6292	66	38.7740
BIC	polar+norm polar	0.5168	86	103.4618
DIC	polar+norm polar	0.5168	86	280.9912
:--:	:--------------:	:-----:	:-----:	:---------:
CV	pol+norm+delta	0.4831	92	50.5893
BIC	pol+norm+delta	0.4382	100	123.5681
DIC	pol+norm+delta	0.4606	96	300.9963
:--:	:--------------:	:-----:	:-----:	:---------:
CV	f_polar	0.4943	90	64.2678
BIC	f_polar	0.4326	101	138.6669
DIC	f_polar	0.4438	99	324.5010

4.Let's try to modify the features_custom taking into account the result of p.3 research:

- firstly include all deltas for cartesian, ground and polar
- secondly include features pol+norm+delta
- thirdly include full f_polar with all polar features.

  As a result - we've got WER = 0.4157 with Correct = 104! So, the leader of the competition is BIC selector with the features_custom = ['delta-norm-rx', 'delta-norm-ry', 'delta-norm-lx', 'delta-norm-ly', 'delta-norm-grnd-rx', 'delta-norm-grnd-ry', 'delta-norm-grnd-lx', 'delta-norm-grnd-ly', 'delta-norm-polar-rr', 'delta-norm-polar-rtheta', 'delta-norm-polar-lr', 'delta-norm-polar-ltheta', 'polar-rr', 'polar-rtheta', 'polar-lr', 'polar-ltheta', 'norm-polar-rr', 'norm-polar-rtheta', 'norm-polar-lr', 'norm-polar-ltheta', 'delta-polar-rr', 'delta-polar-rtheta', 'delta-polar-lr', 'delta-polar-ltheta', 'polar-rr', 'polar-rtheta', 'polar-lr', 'polar-ltheta', 'norm-polar-rr', 'norm-polar-rtheta', 'norm-polar-lr', 'norm-polar-ltheta', 'delta-polar-rr', 'delta-polar-rtheta', 'delta-polar-lr', 'delta-polar-ltheta', 'delta-norm-polar-rr', 'delta-norm-polar-rtheta', 'delta-norm-polar-lr', 'delta-norm-polar-ltheta']

5.Interesting moment - we have to add features into the list with accordance to the ascending Correct meanings after all delta features for minimizing errors from acceleration and deceleration. In this case the WER result will be the same. But if we'd change the order or exclude some other features - the result will be worse, e.g. with features_custom1 the WER = 0.4494 with Correct = 98, which is also not so bad.

6.The full list of features is going to be too noisy, so result is not so good comparing with the features_custom: WER = 0.5337 with Correct = 83.

Feature Reduction. Obviously, the high dimensional appearance-based feature vectors encode a lot of background noise and one would need many more observations to train a robust model. To reduce the number of features and noise and thus the number of parameters to be learned in the models, we apply linear feature reduction techniques to the data. The best obtained result with LDA is 36% WER, whereas with PCA a WER of 27.5% can be obtained. Although theoreticaly LDA should be better suited for pattern recognition tasks, here the training data is insufficient for a numerically stable estimation of the LDA transformation and thus PCA, which is reported to be more stable for high dimensional data with small training sets outperforms LDA [1].

Approaches to adaptation:[2]

1. Model based:
   - Adapt the parameters of the acoustic models to better match the observed data
   - Maximum a posteriori (MAP) adaptation of HMM/GMM parameters
   - Maximum likelihood linear regression (MLLR) of Gaussian parameters
   - Learning Hidden Unit Contributions (LHUC) for neural networks
2. Speaker normalization:
   - Normalize the acoustic data to reduce mismatch with the acoustic models
   - Vocal Tract Length Normalization (VTLN)
   - Constrained MLLR (cMLLR) — model-based normalisation!
3. Speaker space:
   - Estimate multiple sets of acoustic models, characterizing new speakers in terms of these model sets
   - Cluster-adapative training
   - Eigenvoices
   - Speaker codes

I may add that for improving WER is that words are influenced by their context (words before and after). For example is highly likely to see the word Union after the word Soviet.

[1]Speech Recognition Techniques for a Sign Language Recognition System, Philippe Dreuw et al https://www-i6.informatik.rwth-aachen.de/publications/download/154/Dreuw--2007.pdf

[2]Steve Renals Automatic Speech Recognition – ASR Lecture 10 https://www.inf.ed.ac.uk/teaching/courses/asr/2016-17/asr08-adapt.pdf