Hidden Markov Models (HMMs)

Authors: Carleton Smith

Project Guide

- Project Overview
- Part 1: Acquire, Explore, and Preprocess Data
- Part 2: Components of a Hidden Markov Model
- Part 3: Estimating a Hidden Markov Model

Project Overview

EXPECTED TIME: 3 HRS

- Project Overview
- Part 1: Exploring and discretizing data
- Part 2: Components of a Hidden Markov Model
- Part 3: Training a Hidden Markov Model

This will include:

- Clustering
- Deciding the on the type of HMM problem
- Initializing and tuning HMM parameters

Motivation: HMMs can be effective in solving problems in reinforcement learning and temporal pattern recognition.

Objectives: By the end of this assignment, you will:

- Understand the three problems HMMs can solve
- Be able to learn a HMM parameters
- Make predictions using a trained HMM

Problem: Predict the weekly price of corn.

Dataset: *Weekly Corn Prices* from the Kaggle, courtesy of Nick Wong.

Dataset description as provided on Kaggle:

Content
The file composed of simply 2 columns. One is the date (weekend) and the other is corn close price. The period is from 2015-01-04 to 2017-10-01. The original data is downloaded from Quantopian corn futures price.

Inspiration
William Gann: Time is the most important factor in determining market movements and by studying past price records you will be able to prove to yourself history does repeat and by knowing the past you can tell the future. There is a definite relation between price and time.

Please see the Data Folder to explore the data files further.

Note on this project

The implementation of a HMM from scratch can involve complex code that you might not entirely understand. By no means is this implementation intended to be the most robust and efficient. The assignment prioritizes the demonstation of the algorithm's steps while maintaining reproducibility. As such, certain tasks and features non-essential to the algorithm itself (ex: `Signature` and `Parameter` classes) are defined and implemented without explanation. Students are not expected to understand the entirety of the code base. Rather, students will be tested on their high level understanding of HMM and some critical functions.

```
# Import libraries
import numpy as np
import pandas as pd
from collections import namedtuple
from itertools import permutations, chain, product
from sklearn.cluster import KMeans
from inspect import Signature, Parameter
```

```
# Define path constants
ROOT_DIR = '../resource/asnlib/publicdata/'
CORN_2013_2017 = 'corn2013-2017.txt'
CORN_2015_2017 = 'corn2015-2017.txt'
OHL = 'corn_OHLC2013-2017.txt'
```

Part 1: Acquire, Explore, and Preprocess Data

Acquire and Explore

```
corn13_17 = pd.read_csv(ROOT_DIR+CORN_2013_2017, names = ("week","price") )
corn15_17 = pd.read_csv(ROOT_DIR+CORN_2015_2017, names = ("week","price"))
```

```
OHL_df = pd.read_csv(ROOT_DIR+OHL, names = ("week","open","high","low","close"))
```

We will be using just one of these data sets for the purposes of this assignment: `corn13_17`. Students should feel free to experiment on their own using the other datasets provided.

```
corn13_17.head()
```

Inspect the data for missing values.

```
corn13_17.info()
```

The data appear to be complete.

Preprocess: Discretization

As mention in Lecture 11-1, there are two types of HMM:

1. Discrete HMM
2. Continuous HMM

The typical structure of a HMM involves a discrete number of latent ("hidden") states that are unobserved. The observations, which in our case are corn prices, are generated from a state dependent "emission" distribution. Emissions are synonymous with observations.

In the discrete HMM case, the emissions are discrete values. Conversely, the continuous HMM outputs continuous emissions that are generated from the state dependent distribution, which is usually assumed to be Gaussian.

Question 1

```
### GRADED
### TRUE or FALSE: The number of discrete states is a hyperparameter of a HMM.
### Assign you answer as a boolean to ans1

### YOUR ANSWER BELOW
ans1 = True
```

One use case for clustering is to discretize continuous HMM emissions in order to simplify the problem. We will do that in the next section.

Generate Clusters

As noted in Lecture 11-4, clustering a sequence of continuous observations is a form of data quantization. This can simplify the learning of an HMM. Instead of calculating posterior probabilities from a continuous emissions sequence, the observation's respective cluster label is used as the observation. Thus, the emission probability matrix can be encoded as a discrete vector of probabilities.

Question 2

```
### GRADED
### Create a function called `generate_cluster_assignments`
### The function should take 2 parameters:
###     1) pandas Series
###     2) number of clusters
###
### The function should instantiate a sklearn KMeans class
### with the specified number of clusters and a random_state=24.
###
### The function should return a pandas Series of cluster labels for each
### observation in the sequence.
###
### A KMeans object can be instantiated via:
### clusterer = KMeans(args)
###
### NB: That KMeans object has `.fit()` and `.predict()` methods
###
###
### EX:
###
### data_series = pd.Series([1,2,3,2,1,2,3,2,1,2,3,2,1,2,3,2,1,6,7,
###                          8,7,6,7,8,6,7,6,7,8,7,7,8,56,57,58,59,57,58,6,7,8,1,2])
### labels = generate_cluster_assignments(data_series, clusters = 3)
###
### labels --> array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2,
###                   2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 2, 2, 2, 1, 1])

### NOTE: Your particular labels might not match exactly, but the CLUSTERS should be the same


### YOUR ANSWERS BELOW
def generate_cluster_assignments(ser, clusters):
    cluster = KMeans(n_clusters = clusters, random_state=24)
    df = pd.DataFrame(series)
    cluster.fit(df)

    return cluster.predict(df)
```

```
# Cluster 2013-2017
corn13_17_seq = generate_cluster_assignments(corn13_17[['price']], 5)
```

Part 2: Components of a Hidden Markov Model

HMM Parameters

A HMM consists of 5 components:

- N -- The number of hidden states
- A -- State transition matrix
- B -- Emission probability matrix
- π -- Starting likelihood
- $(x_{i}, …, x_{T})$ -- Sequence of emissions, or observations

N: Number of hidden states - this is a discrete integer that the practitioner provides. This is the number of assumed hidden states.

A: State transition matrix - With N==2, the transition matrix may look like the following

- $\begin{bmatrix}
  . & S1 & S2 \
 S1 & .7 & .3 \
 S2 & .6 & .4
  \end{bmatrix}$
- The first row shows the probability of transitioning from state 1 --> (state 1=0.7, state 2=0.3)
- The second row shows the probability of transitioning from state 2 --> (state 1=0.6, state 2=0.4).

B: Emission probability Matrix - With M==4 (number of unique observations)

- $\begin{bmatrix}
  . &a & b & c & d \\
  S1 & .4 & .3 & .1 & .2 \
 S2 & .1 & .4 & .1 & .4
  \end{bmatrix}$
- The rows correspond to state 1 and state 2, respectively. However, the columns in B correspond to the probability of that observation, given the respective state.
- For example, the probability of observing d in $S1$ == 0.2

π: Initial state probabilities

- π = $ \begin{bmatrix}
  .5 \
  .5
  \end{bmatrix}$
- This says the sequence of states is equally likely to start in $S1$ or $S2$

$(x_{i}, …, x_{T})$: This is the observed sequence.

The only component of an HMM that will always be known is the sequence of observations, $(x_{i}, …, x_{T})$. The type of HMM problem is determined by which of these components are known and the motivation of the problem. We will discuss the types of HMM problems next.

Question 3

```
### GRADED
### Multiple Choice:
### If `N` is the number of states, the shape of "A" (The transition matrix) will always be:
### a: 2 x 2
### b: N x N
### c: N x number of unique observations
### d: None of the above
###
### Assign the character associated with your choice as a string to to `ans4`

### YOUR ANSWERS BELOW
ans4 = 'b'
```

Question 4

```
### GRADED
### Multiple Choice:
### If `N` is the number of states, The shape of "B" (emission matrix) will always be:
### a: 2 x 4
### b: N x N
### c: N x number of unique observations
### d: None of the above
###
### assign your answer as a string to `ans5`

### YOUR ANSWERS BELOW
ans5 = 'c'
```

Types of Problems using HMM

In Lecture 11-2, three HMM estimation problems are described.

1. State Estimation
2. State Sequence
3. Learn an HMM

We will briefly cover the motivation of each estimation problem.

1. State Estimation:

- Given an HMM (π, *A*, *B*), and an observation sequence $(x_{i}, …, x_{T})$
- Estimate the state probability for x_{i}

2. State Sequence:

- Given an HMM (π, *A*, *B*), and an observation sequence $(x_{i}, …, x_{T})$
- Estimate the most probable state sequence

3. Learn an HMM:

- Given an observation sequence $(x_{i}, …, x_{T})$
- Estimate the HMM parameters (π, *A*, *B*)

Question 5

```
### GRADED
### Multiple Choice: Which of the following HMM problems
### use the Forward-Backward Algorithm to estimate the solution?
###
### HINT: This is covered in 11-2
###
### a: State Estimation
### b: State Sequence
### c: Learn an HMM
### d: None of the above
### e: All of the above
###
### list all that apply as 'a', 'b', 'c', 'd', and/or 'e' in
### the list assigned to ans6

### YOUR ANSWERS BELOW
ans6 = ['a','c']
```

Question 6

```
### GRADED
### Multiple Choice: Which of the following HMM problems
### use the Viterbi Algorithm to estimate the solution?
###
### HINT: This is covered in 11-2
###
### a: State Estimation
### b: State Sequence
### c: Learn an HMM
### d: None of the above
### e: All of the above
###
### list all that apply as 'a', 'b', 'c', 'd', and/or 'e' in
### the list assigned to ans7

### YOUR ANSWERS BELOW
ans7 = ['b']
```

Part 3: Estimating a Hidden Markov Model

In this section, you will be guided through an exercise of the third HMM estimation problem: Learn an HMM. We will use the the Corn Prices 2013-2017 dataset from Kaggle as our sequence of observations.

Question 3 of Part 1 asked you to make a function that will quantize a Pandas Series into a specified number of clusters. We will use this function now to discretize our price data into 5 clusters.

```
# Cluster 2013-2017
corn13_17_seq = generate_cluster_assignments(corn13_17[['price']], 5)
corn13_17_seq
```

NOTE: Quite a few functions are provided in the following cells. It is not imperative that you completely understand how each of them work. Many of them are helper functions that perform a specific task within another function. What is important is that you understand the procedure that is occuring to estimate the HMM parameters. Those steps are laid out in the next section.

Steps for Learning a HMM

The Expectation Maximization (EM) algorithm is used to estimate the parameters of a HMM given a sequence of observations (aka emissions). Here are the general steps for this procedure:

1. Initialize a set of parameters for the HMM (π, *A*, *B*)
2. Conduct the EM algorithm:
    - The E Step: Use forward-backward algorithm to calculate the probability of observing the emissions with the given HMM parameters (π, *A*, *B*)
    - The M Step: Update the HMM parameters so that the sequence of observations are more likely to have come from this particular HMM
3. Repeat steps 1 and 2 until the HMM parameters have converged.

The remaining parts of this assignment will perform this procedure.

Important Constant: Number of Unique States

```
# Almost all functions require this constant as an argument
STATE_LIST = ['S1', 'S2']
```

```
# Initialze state transition probabilities (2 states)
STATE_TRANS_PROBS = [0.4, 0.6, 0.35, 0.55]
```

Helper Functions

These functions are used to perform common tasks.

```
# given a list with unique items, this function will return a new list with all permutations
def make_state_permutations(list_of_unique_states):
    l1 = [''.join(tup) for tup in permutations(list_of_unique_states, 2)]
    l2 = [state+state for state in list_of_unique_states]
    return sorted(l1 + l2)

# helper function in EM function
def _grab_highest_prob_and_state(state_permutations_lst, prob_arr):
    return (prob_arr[np.argmax(prob_arr)], state_permutations_lst[np.argmax(prob_arr)])
```

The following two functions transform a dictionary to a different format.

```
def dict_to_tuples(list_of_unique_states, d):
    """
    list_of_unique_states: List of unique state names, as strings
    d: Dictionary of state transition probabilities


    EXAMPLE:
    s_perms = ['S1S1', 'S1S2', 'S2S1', 'S2S2']
    p_list = [0.1, 0.9, 0.4, 0.6]
    d = {'S1S1': 0.1, 'S1S2': 0.9, 'S2S1': 0.4, 'S2S2': 0.6}

    print(dict_to_tuples(d))

    OUTPUT:
    {S1: (0.1, 0.9), S2: (0.4, 0.6)}
    """

    # Defensive programming to ensure output will be correct
    list_of_unique_states = sorted(list_of_unique_states)
    assert make_state_permutations(list_of_unique_states) == list(d.keys()), \
            "Keys of dictionary must match output of `make_state_permutations(list_of_unique_states)`"

    lengths = [len(st) for st in list_of_unique_states]
    final_dict = {}
    for idx, st in enumerate(list_of_unique_states):
        tup = []
        for trans_p in d.keys():
            if trans_p[:lengths[idx]] == st:
                tup.append(d[trans_p])
            else:
                continue
        final_dict[st] = tuple(tup)

    return final_dict
```

```
def obs_to_tuples(list_of_unique_states, d, sequence):
    """
    list_of_unique_states: List of unique state names, as strings
    d: Dictionary of obs transition probabilities
    sequence: the observation sequence


    EXAMPLE:
    STATE_LIST = ['S1', 'S2']
    d = {'S1_0': 0.1,
         'S1_1': 0.3,
         'S1_2': 0.4,
         'S1_3': 0.15,
         'S1_4': 0.05,
         'S2_0': 0.15,
         'S2_1': 0.2,
         'S2_2': 0.3,
         'S2_3': 0.05,
         'S2_4': 0.3}
    corn15_17_seq = generate_cluster_assignments(corn15_17[['price']], 5)

    print(obs_to_tuples(STATE_LIST, d))

    OUTPUT:
    {'S1': (0.1, 0.3, 0.4, 0.15, 0.05), 'S2': (0.15, 0.2, 0.3, 0.05, 0.3)}
    """

    # Defensive programming to ensure output will be correct
    list_of_unique_states = sorted(list_of_unique_states)
    num_unique_obs = len(np.unique(sequence))

    lengths = [len(st) for st in list_of_unique_states]
    final_dict = {}
    for idx, st in enumerate(list_of_unique_states):
        tup = []
        for e_trans in d.keys():
            if e_trans[:lengths[idx]] == st:
                tup.append(d[e_trans])
            else:
                continue
        final_dict[st] = tuple(tup)

    return final_dict
```

Define Transition Functions

The following three function definitions (all starting with `generate`) will be used to create our initial HMM parameters (π, A, B) in the form of a dictionary.

The functions are flexible enough to create (π, A, B) from user specified values, or provide default uniform probability vectors if no values are explicitly given in the `**kwargs` argument.

An Aside: Why dictionaries and not arrays?

Dictionaries were the selected as the data structure for this exercise instead of arrays for efficiency and accuracy. The functions defined for this procedure involve retrieving values from data structures frequently and altering existing values. This is best done in a dictionary vs an array because dictionaries utilize hashing functions to look up data instead of indexed positions.

To demonstate the speed benefits of using dictionaries to retrieve values, consider the time needed to retrieve a piece of data from both an array and a dictionary in the following exercise.

```
import timeit

all_setup = """

import numpy as np

# These hold the same information
arr = np.array([[0.4, 0.6], [0.35, 0.55]])
d = {'S1S1': 0.4, 'S1S2': 0.6, 'S2S1': 0.35, 'S2S2': 0.55}

"""
i = 10_000_000
index_an_array = 'arr[0,0]'
retrieve_value = "d['S1S1']"

print('Seconds to index an array {} times: {}'.format(
    i, timeit.timeit(setup=all_setup, stmt=index_an_array, number=i)))

print('\n','#' * 60, '\n')
print('Seconds to retrieve value {} times: {}'.format(i, timeit.timeit(setup=all_setup, stmt=retrieve_value, number=i)))
```

The following functions produce initial probabilities for (π, A, B)

```
def generate_state_trans_dict(list_of_unique_states, **kwargs):
    '''

    'list_of_unique_states': list of states as strings
    ''**kwargs': keyword being the state and value is tuple of state transitions.
    <Must be listed in same order as listed in 'list_of_unique_states'>

    If **kwargs omitted, transitions are given uniform distribution based on
    number of states.

    EXAMPLE1:
    state_params = generate_state_trans_dict(['S1', 'S2', 'S3'])

    OUTPUT1:
    {'S1S1': 0.5, 'S2S2': 0.5, 'S1S2': 0.5, 'S2S1': 0.5}

    EXAMPLE2:
    state_params = generate_state_trans_dict(['S1', 'S2'], S1=(0.1, 0.9), S2=(0.4, 0.6))

    OUTPUT2:
    {'S1S1': 0.1, 'S1S2': 0.9, 'S2S1': 0.4, 'S2S2': 0.6}

    '''
    # number of states
    N = len(list_of_unique_states)

    # this runs if specific transitions are provided
    if kwargs:
        state_perms = [''.join(tup) for tup in permutations(list(kwargs.keys()), 2)]
        all_permutations = [state+state for state in list_of_unique_states] + state_perms
        pbs = chain.from_iterable(kwargs.values())
        state_trans_dict = {perm:p for perm, p in zip(sorted(all_permutations), pbs)}
        return state_trans_dict

    state_perms = [''.join(tup) for tup in permutations(list_of_unique_states, 2)]
    all_permutations = [state+state for state in list_of_unique_states] + state_perms
    state_trans_dict = {perm: (1/N) for perm in all_permutations}
    return state_trans_dict
```

```
def generate_emission_prob_dist(list_of_unique_states, sequence, **kwargs):
    '''
    list_of_unique_states: list of states as strings
    sequence: array of observations

    EXAMPLE1:
    corn15_17_seq = generate_cluster_assignments(corn15_17[['price']])
    STATE_LIST = ['S1', 'S2']

    generate_emission_prob_dist(STATE_LIST, corn15_17_seq, S1=(0.1, 0.3, 0.4, 0.15, 0.05))

    OUTPUT1:
    {'S1_0': 0.1,
     'S1_1': 0.3,
     'S1_2': 0.4,
     'S1_3': 0.05,
     'S1_4': 0.05,
     'S2_0': 0.2,
     'S2_1': 0.2,
     'S2_2': 0.2,
     'S2_3': 0.2,
     'S2_4': 0.2}
    '''
    # number of unique obs
    B = list(np.unique(sequence).astype(str))

    # this runs if specific transitions are provided
    if kwargs:
```

```
        for t in kwargs.values():
            assert len(t) == len(B), "Must provide all probabilities for unique emissions in given state."
            assert round(np.sum(t)) == 1.0, "Given emission probabilities for a state must add up to 1.0"
        for k in kwargs.keys():
            assert k in list_of_unique_states, "Keyword arguments must match a value included in `list_of_unique_states`"
        diff = list(set(list_of_unique_states).difference(kwargs.keys()))

        pbs = chain.from_iterable(kwargs.values())
        obs_perms = [state + '_' + str(obs) for state in kwargs.keys() for obs in B]

        obs_trans_dict = {perm:p for perm, p in zip(sorted(obs_perms), pbs)}

        if diff:
            obs_perms_diff = [state + '_' + obs for state in diff for obs in B]
            obs_trans_dict.update({perm: (1/len(B)) for perm in obs_perms_diff})

        return obs_trans_dict

    obs_perms = [state + '_' + obs for state in list_of_unique_states for obs in B]
    obs_trans_dict = {perm: (1/len(B)) for perm in obs_perms}
    return obs_trans_dict
```

```
def generate_init_prob_dist(list_of_unique_states, **kwargs):
    """
    Examples:
    STATE_LIST = ['S0','S1','S2','S3','S4']
    initial_states = {'S1':.2, 'S2':.3, 'S3':.05, 'S4':.25, 'S0':.2}

    print(generate_init_prob_dist(STATE_LIST))
    # --> {'S0': 0.2, 'S1': 0.2, 'S2': 0.2, 'S3': 0.2, 'S4': 0.2}

    print(generate_init_prob_dist(STATE_LIST, **initial_states))  ### NOTE: must unpack dictionary with **
    # --> {'S1': 0.2, 'S2': 0.3, 'S3': 0.05, 'S4': 0.25, 'S0': 0.2}
    """

    # number of states
    N = len(list_of_unique_states)

    # this runs if specific transitions are provided
    if kwargs:
        for t in kwargs.values():
            assert isinstance(t, float), "Must provide probabilities as floats."
            assert t > 0, "Probabilities must be greater than 0."
        assert np.sum(list(kwargs.values())) == 1.0, "Given probabilities must add up to 1.0"
        assert len(kwargs) == len(list_of_unique_states), "Please provide initial probabilities for all states, or leave blank"

        # build the prob dictionary
        init_prob_dict = {item[0]: item[1] for item in kwargs.items()}
        return init_prob_dict

    init_prob_dist = {state: (1/N) for state in list_of_unique_states}
    return init_prob_dist
```

Create State Transition Priors

```
# Make permutations of state transition (this len should match len(STATE_TRANS_PROBS))
state_transitions_list = make_state_permutations(STATE_LIST)

# Create transition matrix in form of dictionary
state_transition_probs = {
    trans: prob for trans, prob in zip(state_transitions_list, STATE_TRANS_PROBS)
}

state_transition_probs
```

The dictionary shown above is the state transition "matrix" formated as a dictionary. This dictonary can be represented in another format as well, which will be convenient when provided as a `**kwargs` argument. This transformation occurs with the function `dict_to_tuples` in the cell below.

```
# Transform dictionary to be in tuple format
#### - this format is required in the `generate_*` functions used later
A_prior = dict_to_tuples(STATE_LIST, state_transition_probs)
A_prior
```

NOTE ON FORMATTING

Some functions in this assignment require the transition probabilities be in a specific format. We will switch between two formats that hold the same information:

With 2 states:

Format 1: `{'S1S1': 0.4, 'S1S2': 0.6, 'S2S1': 0.35, 'S2S2': 0.55}`

The above dictionary is the state transition matrix in dictionary form, where every key is a transition from one state to another. For example, the key-value pair, `'S1S2': 0.6`, says the probability of the state moving from `S1` to `S2` from observation `i` to observation `j` is `0.6`.

Format 2: `{'S1': (0.4, 0.6), 'S2': (0.35, 0.55)}`

The second format contains the same information as the first, but encodes the probabilities in tuples. With only two states and assuming `S1` is the first state, `'S1': (0.4, 0.6)` is interpreted as the probability of staying in `S1` is 0.4 and the probability of moving from `S1` to `S2` is 0.6.

Create Emission Probability Priors

```
# Manually initialize emission probabilities - in format 1
B_format1 = {
    'S1_0': 0.1,
    'S1_1': 0.3,
    'S1_2': 0.4,
    'S1_3': 0.15,
    'S1_4': 0.05,
    'S2_0': 0.15,
    'S2_1': 0.2,
    'S2_2': 0.3,
    'S2_3': 0.05,
    'S2_4': 0.3
}
```

The emission probabilities can be stored in dictionary format as well. Using the function `obs_to_tuples()` function in the cell below, we convert the emission probabilities to a dictionary format that is well suited to be provided as an argument in `**kwargs`.

```
# Convert emission matrix to format 2
B_format2 = obs_to_tuples(STATE_LIST, B_format1, corn13_17_seq)
B_format2
```

The emission probabilities can be converted back to the format of `B_format1` using the previously defined `generate_emission_prob_dist` function. This is demonstrated in the following cell:

```
# Use `generate_emission_prob_dist` to convert B back to format 1
generate_emission_prob_dist(STATE_LIST, corn13_17_seq, **B_format2)
```

We will keep the emission probabilities in the `"key": tuple` format so that it can be used easily as a `**kwargs` argument later.

```
B_prior = obs_to_tuples(STATE_LIST, B_format1, corn13_17_seq)
```

Let's recap.

A fair amount of setup has already occurred and we have not yet started the HMM Learning procedure. Let's take a moment to recap the important elements we have established so far.

We have a state transition matrix, A (prior for A)

```
A_prior
```

We have an emissions probability matrix, B (prior for B)

```
B_prior
```

We need an initial state probability matrix, π. We will use `generate_init_prob_dist` to do this.

We can also use the `generate_init_prob_dist` function without specified parameters to make uniform initial state probabilities.

Note: If `pi__init` is not provided, a uniform distribution is produced based on number of states.

```
# User specified initial probabilities
pi__init = {'S1': 0.4 , 'S2': 0.6}

# generate the dictionary holding initial state probabilities
pi = generate_init_prob_dist(STATE_LIST, **pi__init)
pi
```

```
# Using default initial parameters - demonstration only, we won't save this dictionary
generate_init_prob_dist(STATE_LIST)
```

And we have defined a number of functions that will be involved in the EM algorithm in some way

Constants:
STATE_LIST
STATE_TRANS_PROBS

Functions:
`generate_cluster_assignments`
`make_state_permutations`
`_grab_highest_prob_and_state`
`dict_to_tuples`
`obs_to_tuples`
`generate_state_trans_dict`
`generate_emission_prob_dist`
`generate_init_prob_dist`

Finally, we need to create a data structure that will hold all of our probability calculations until we are finished computing EStep

For this task, we will take advantage of a powerful data structure from the `collections` module: `namedtuple`.

NAMED TUPLES

Take a few minutes to review the documentation on `namedtuples`. Then answer the following question.

Alternatively, for a short and helpful introduction, review this tutorial.

Question 7

```
# Consider the following array of probabilities:
probs = np.array([0.3, 0.7])
```

```
### GRADED

###########
### NOTE: This question has a very low point-total; it will not impact your ability
### to pass this assignment. It should be considered something of an
### optional extension.
###########

### Create a namedtuple factory called 'State' that
### has two field names: `prob1` and `prob2`.
###
### After defining the factory, instantiate an instance
### of the 'State' factory called `s1` and store the two probabilities
### contained in the array `probs` (defined above) to
### the `prob1` and `prob2` field names, respectively.
###
### assign the value of the `prob1` field name to `ans8` below.
###
### EXAMPLE (after creating and instantiating):
### print(s1.prob1)  --> 0.3
### ans8 = s1.prob1


### YOUR ANSWERS BELOW
State = namedtuple('State', 'prob1 prob2')
s1 = State(prob1=0.3, prob2=0.7)
ans8 = s1.prob1
```

Putting it all together

We now have all the pieces to use the EM algorithm. To do this requires the calculation of forward backward algorithm. We will use what we recently learned about `namedtuple` from the `collections` module to do this.

The function below will generate this data structure for us.

```
def generate_obs_data_structure(sequence):
    #  sequence: 1D numpy array of observations
    ObservationData = namedtuple(
        'ObservationData',
        ['prob_lst', 'highest_prob', 'highest_state']
    )
    return {index+1: ObservationData for index in np.arange(len(sequence)-1)}
```

STEP 1: ESTIMATE PROBABILITIES:

This step involves using the Forward-Backward algorithm to calculate the probability of observing a sequence, given a set of HMM parameters. We have all the tools to do this now.

```
# Enforce an Argument Signature on following function to prevent errors with **kwargs
params = [Parameter('list_of_unique_states', Parameter.POSITIONAL_OR_KEYWORD),
         Parameter('sequence', Parameter.POSITIONAL_OR_KEYWORD),
         Parameter('A', Parameter.KEYWORD_ONLY, default=generate_state_trans_dict),
         Parameter('B', Parameter.KEYWORD_ONLY, default=generate_emission_prob_dist),
         Parameter('pi', Parameter.KEYWORD_ONLY, default=generate_init_prob_dist)]

sig = Signature(params)

def calculate_probabilities(list_of_unique_states, sequence, **kwargs):

    # enforce signature to ensure variable names
    bound_values = sig.bind(list_of_unique_states, sequence, **kwargs)
    bound_values.apply_defaults()


    # grab params that are left to default values
    param_defaults = [(name, val) for name, val in bound_values.arguments.items() if callable(val)]

    # grab non-default params
    set_params = [(name, val) for name, val in bound_values.arguments.items() if isinstance(val, dict)]

    # this will run if any default hmm parameters are used
    if param_defaults:
        for name, val in param_defaults:
            if name == 'B':
                B = val(list_of_unique_states, sequence)
            elif name == 'A':
                A = val(list_of_unique_states)
            elif name == 'pi':
                pi = val(list_of_unique_states)
            else:
                continue

    # this will run if kwargs are provided
```

```
    if set_params:
        for name, val in set_params:
            if name == 'B':
                B = generate_emission_prob_dist(list_of_unique_states, sequence, **val)
            elif name == 'A':
                A = generate_state_trans_dict(list_of_unique_states, **val)
            elif name == 'pi':
                pi = generate_init_prob_dist(list_of_unique_states, **val)
            else:
                continue

    # instantiate the data structure
    obs_probs = generate_obs_data_structure(sequence)

    # all state transitions
    state_perms = make_state_permutations(list_of_unique_states)

    # for every transition from one observation to the next, calculate probability of going from Si to Sj
    # loop through observations
    for idx, obs in enumerate(sequence):

        if idx != 0:  # check if this is the first observation
            # instantiate the namedtuple for this observation
            obs_probs[idx] = obs_probs[idx]([], [], [])

            # loop through each possible state transition
            for st in state_perms:

                # calculate prob of current obs for this state
                prev_prob = pi[st[:2]] * B[st[:2]+'_'+str(sequence[idx-1])]

                # calculate prob of previous obs for this state
                curr_prob = A[st] * B[st[2:]+'_'+str(obs)]

                # combine these two probabilities
                combined_prob = round(curr_prob * prev_prob, 4)

                # append probability to the list in namedtuple
                obs_probs[idx].prob_lst.append(combined_prob)

            # check for highest prob of observing that sequence
            prob_and_state = _grab_highest_prob_and_state(state_perms, obs_probs[idx].prob_lst)
            obs_probs[idx].highest_prob.append(prob_and_state[0])
            obs_probs[idx].highest_state.append(prob_and_state[1])

        else: # this is the first observation, exit loop.
            continue
    return (obs_probs, A, B, pi)
```

```
ob_prob, A, B, pi = calculate_probabilities(STATE_LIST, corn13_17_seq, A=A_prior, B=B_prior, pi=pi)
```

```
ob_prob
```

```
A
```

```
B
```

```
pi
```

STEP 2: UPDATE PARAMETERS

Update the State Transition Matrix

```
# This function sums all of the probabilities and
# outputs a new (un-normalized) state transition matrix
def new_state_trans(STATE_LIST, probabilities):
    state_perms = make_state_permutations(STATE_LIST)
    sums_of_st_trans_prob = {p:0 for p in state_perms}
    highest_prob_sum = 0
    for obs in probabilities:
        highest_prob_sum += probabilities[obs].highest_prob[0]
        for i, p in enumerate(sums_of_st_trans_prob):
            sums_of_st_trans_prob[p] += probabilities[obs].prob_lst[i]

    for key in sums_of_st_trans_prob:
        sums_of_st_trans_prob[key] = sums_of_st_trans_prob[key] / highest_prob_sum

    # finally, normalize so the rows add up to 1
    for s in STATE_LIST:
        l = []
        for k in sums_of_st_trans_prob:
            if s == k[:2]:
                l.append(sums_of_st_trans_prob[k])
        for k in sums_of_st_trans_prob:
            if s == k[:2]:
                sums_of_st_trans_prob[k] = sums_of_st_trans_prob[k] / sum(l)

    return sums_of_st_trans_prob
```

```
# Update and normalize posterior state transition
A_posterior = new_state_trans(STATE_LIST, ob_prob)
```

```
A_posterior
```

```
# Convert state transition to "format 2" so it can be
# used as input in the next iteration of "E" step
A_posterior = dict_to_tuples(STATE_LIST, A_posterior)
```

Update the Emission Probabilities

Here, we define some functions designed to do specific tasks.

```
##### tally up all observed sequences
def observed_pairs(sequence):
    observed_pairs = []
    for idx in range(len(sequence)-1):
        observed_pairs.append((sequence[idx], sequence[idx+1]))
    return observed_pairs
```

```
def make_emission_permutations(sequence):
    unique_e = np.unique(sequence)
    return list(product(unique_e, repeat = 2))

make_emission_permutations([1,1,0, 2])
make_emission_permutations([0,1,0,3,0])
```

```
def find_highest_with_state_obs(prob_pairs, state, obs):
    for pp in prob_pairs:
        if pp[0].count((state,obs))>0:
            return pp[1]
```

```
def normalize_emissions(b_tuple_format):
    new_b_dict = {}
    for key, val in b_tuple_format.items():
        denominator = sum(val)
        new_lst = [v/denominator for v in val]
        new_b_dict[key] = tuple(new_lst)
    return new_b_dict
```

Finally, we are ready to update the emission probabilities with the function below

```
def emission_matrix_update(sequence, state_list, A, B, pi):
    state_pairs = list(product(state_list, repeat = 2))
    obs_pairs = observed_pairs(sequence)

    new_B = {}
    for obs in np.unique(sequence): # For every unique emission

        # Find all the sequence-pairs that include that emission
        inc_seq = [seq for seq in obs_pairs if seq.count(obs)>0]

        # Collector for highest-probabilities
        highest_pairs = []

        # For each sequence-pair that include that emission
        for seq in inc_seq:

            prob_pairs = []

            # Go through each potential pair of states
            for state_pair in state_pairs:

                state1, state2 = state_pair
                obs1, obs2 = seq

                # Match each state with it's emission
                assoc_tuples = [(state1, obs1),
                                (state2, obs2)]

                # Calculate the probability of the sequence from state
                prob = pi[state1] * B[state1+"_"+str(obs1)]
                prob *= A[state1+state2]*B[state2+"_"+str(obs2)]
                prob = round(prob,5)
                # Append the state emission tuples and probability
                prob_pairs.append([assoc_tuples, prob])

            # Sort probabilities by maximum probability
            prob_pairs = sorted(prob_pairs, key = lambda x: x[1], reverse = True)

            # Save the highest probability
            to_add = {'highest':prob_pairs[0][1]}
            # Find the highest probability where each state is associated
            # With the current emission
            for state in STATE_LIST:

                highest_of_state = 0

                # Go through sorted list, find first (state,observation) tuple
                # save associated probability

                for pp in prob_pairs:
                    if pp[0].count((state,obs))>0:
                        highest_of_state = pp[1]
                        break

                to_add[state] = highest_of_state
```

```
            # Save completed dictionary
            highest_pairs.append(to_add)

        # Total highest_probability
        highest_probability =sum([d['highest'] for d in highest_pairs])

        # Total highest probabilities for each state; divide by highest prob
        # Add to new emission matrix
        for state in STATE_LIST:
            new_B[state+"_"+str(obs)]= sum([d[state] for d in highest_pairs])/highest_probability


    return new_B
```

Run the function:

```
nb = emission_matrix_update(corn13_17_seq,STATE_LIST, A,B,pi)
nb
```

The emission probabilities are updated, but they need to be normalized. To do this, we will convert to dictionary to the `key: tuple` format and normalize so that the probabilities add up to 1.

```
B_ = obs_to_tuples(STATE_LIST, nb, corn13_17_seq)
B_posterior = normalize_emissions(B_)
```

```
# normalized state transition posterior:
A_posterior
```

```
# normalized emission posterior probabilities
B_posterior
```

STEP 3: REPEAT UNTIL PARAMETERS CONVERGE

```
ob_prob2, A2, B2, pi2 = calculate_probabilities(STATE_LIST, corn13_17_seq, A=A_posterior, B=B_posterior, pi=pi)
ob_prob2
```

```
A_post2 = new_state_trans(STATE_LIST, ob_prob2)  # update and normalize state transition matrix again
A_post2 = dict_to_tuples(STATE_LIST, A2)  # convert to `key: tuple` format
A_post2
```

```
# update emissions matrix again
nb2 = emission_matrix_update(corn13_17_seq, STATE_LIST, A2, B2, pi)  # update emissions matrix again
B_post2 = obs_to_tuples(STATE_LIST, nb2, corn13_17_seq)  # convert emission posterior to `key:tuples` format
B_post2 = normalize_emissions(B_post2)  # normalize emissions probabilities
B_post2
```