Recipe name: markov_chain

Inputs:

- data, sequence: a sequence of numbers, each between 0 and 3 inclusive
- order, integer: a number greater than or equal to 1 that will represent the order of the returned markov chain.

Outputs:

- final_chain, map: a correspondence of tuple keys that will map to nested maps

Steps:

1. Initialize final_chain as an empty map, setting the default value to what empty_dict returns
2. Initialize offset as 0
3. While offset < the length of data – order,
    a. Initialize key as an empty sequence
    b. For every number from 0 to order-1,
        i. Find the element of data at the index number+offset and append it to the end of key
    c. Set tuple_key to a tuple representation of key
    d. Take the map that tuple_key maps to in final_chain and iterate the value that the element of data at the index order+offset maps to by 1.0
    e. ***DELETE THIS:*** final_chain[tuple_key][data[order+offset]] += 1.0
    f. Iterate offset by 1
4. For every base_key and base_value in final_chain's keys and values,
    a. Initialize a counter, total, to keep track of how many times each inner map is iterated in total
    b. For every inner_value in the values of the map that base_key corresponds to,
        i. total += inner_value
    c. For every inner_key in the keys of the map that base_key corresponds to,
        i. The value at inner_key of the value at base_key of final_chain /= total
5. return final_chain

Helper recipe name: empty_dict

Inputs:

- None

Outputs:

- an empty map with default values set to 0.0

Steps:

1. Returns an empty map with default values of 0.0

Recipe name: predict

Inputs:

- model, map: a map of tuple keys corresponding to nested maps that represents a markov chain
- last, sequence: a sequence of numbers, each between 0-3, whose length is the same as the model's order
- num, integer: the number of states to predict

Outputs:

- prediction, sequence: a sequence of numbers of length num, with each number being between 0-3

Steps:

```
1. Initialize a map, default, for the situation of the sequence not being in the model, i.e a
   map that will map each number from 0-3 to the value 0.25.
2. Create a copy of last, last_copy, that can be edited during the function runtime
3. Create a new variable, order, equal to the length of last
4. Create a new integer, offset, that will represent how many times the loop has run
5. Initialize prediction as an empty sequence
6. While offset is less than num,
    a. Initialize a new sequence, last_state, as the last order number of digits of
       last_copy
    b. Create a variable last_tuple that will represent last_state as a tuple
    c. If last_tuple is a key in model,
        i. Create a new integer, latest, that will be the result of the weighted_choice
           function with the value that last_tuple corresponds to
    d. Otherwise,
        i. Create a new integer, latest, that will be the result of the weighted_choice
           function with default
    e. Append latest to the end of last_copy
    f. Append latest to the end of prediction
    g. Iterate offset
7. Return prediction
```

Helper recipe name: weighted_choice

Inputs:

- prob_dict - a map with anything(that is possible) as a key and probabilities as the values

Output:

- choice - the chosen key

Steps:

1. Initialize two sequences, keys and probabilities
2. For each key and value in prob_dict,
    a. Append the key to the end of keys
    b. Append the value to the end of probabilities
3. Create a new float, chance, that will be the result of a random number generation from 0-1.0
4. Set a new float, previous, to 0.
5. For each num in the range of probabilities' length,
    a. If chance is between previous and the value at probabilities corresponding to num + previous,
        i. Return the value of keys corresponding to num
    b. Add the value at probabilities corresponding to num to previous

Experiments writeup

Given FSLR’s actual [3, 0, 0, 1, 0],

```
Order 1 : 3.4747999999999994

Order 3 : 3.1023999999999997

Order 5 : 3.3763999999999999

Order 7 : 3.0735999999999999

Order 9 : 3.0691999999999998
```

Given GOOG’s actual [1, 3, 3, 1, 1],

```
Order 1 : 2.1788

Order 3 : 1.4688

Order 5 : 1.9439999999999999

Order 7 : 2.2751999999999999

Order 9 : 2.3579999999999997
```

Given DJIA’s actual [2, 2, 2, 2, 1],

```
Actual: [2, 2, 2, 2, 1]

Order 1 : 0.95039999999999991

Order 3 : 0.92039999999999999

Order 5 : 0.79080000000000001

Order 7 : 1.1548

Order 9 : 1.5232000000000002
```

Discussion

1. The orders that best predicted FSLR, GOOG, and DJIA's numbers were 9, 3, and 5 respectively. I think this is the case because of the deviancy between the numbers of each stock's given tests. FSLR went from 3 to 0, then got a 1 in the middle, so with a higher deviation it makes sense for the lowest error to be from a higher order. GOOG and DJIA, on the other hand, were more consistent in their stocks and so it took less orders to predict more accurately.
2. DJIA has the lowest error. On the plot graph, we can see that the DJIA stock has the most consistent stock with low changes. This makes sense on the histogram as well, as we can see that DJIA has more days in adjacent bins than the other stocks.
3. For an nth order Markov chain, there is a maximum of 4^n states. It's like binary, where every digit increases the number of states exponentially, but instead of there being 2 states there's 4.
4. No, you can only see all the possible states in a markov chain of order 4 or lower. Any higher and there are simply too many states, with a 5th order markov chain having 1024. It's impossible, therefore, to get all 1024 states in a 502-day sample. If there's not enough data then it's a lot more difficult to judge if your model is truly accurate.