Assignment 2

Daniel Mehta

Exercise 1: POS (Part-of-Speech) Tagging with Hidden Markov Model

In [3]:
```
import nltk
import numpy as np
import pandas as pd
import random
from sklearn.model_selection import train_test_split
import pprint, time
```

In [4]:
```
nltk.download('treebank')
nltk.download('universal_tagset')
nltk_data = list(nltk.corpus.treebank.tagged_sents(tagset='universal'))
for sent in nltk_data[:2]:
  for tuple in sent:
    print(tuple)
```

```
[nltk_data] Downloading package treebank to
[nltk_data]     /Users/danielmehta/nltk_data...
[nltk_data]   Package treebank is already up-to-date!
[nltk_data] Downloading package universal_tagset to
[nltk_data]     /Users/danielmehta/nltk_data...
[nltk_data]   Package universal_tagset is already up-to-date!
```

```
('Pierre', 'NOUN')
('Vinken', 'NOUN')
(',', '.')
('61', 'NUM')
('years', 'NOUN')
('old', 'ADJ')
(',', '.')
('will', 'VERB')
('join', 'VERB')
('the', 'DET')
('board', 'NOUN')
('as', 'ADP')
('a', 'DET')
('nonexecutive', 'ADJ')
('director', 'NOUN')
('Nov.', 'NOUN')
('29', 'NUM')
('.', '.')
('Mr.', 'NOUN')
('Vinken', 'NOUN')
('is', 'VERB')
('chairman', 'NOUN')
('of', 'ADP')
('Elsevier', 'NOUN')
('N.V.', 'NOUN')
(',', '.')
('the', 'DET')
('Dutch', 'NOUN')
('publishing', 'VERB')
('group', 'NOUN')
('.', '.')
```

In [5]:

```
#Split training and validation
train_set,test_set =train_test_split(nltk_data,train_size=0.80,test_size=0.20,random_state = 101)
```

In [6]:

```
train_tagged_words = [ tup for sent in train_set for tup in sent ]
test_tagged_words = [ tup for sent in test_set for tup in sent ]
print(len(train_tagged_words))
print(len(test_tagged_words))
```

```
80310
20366
```

In [7]:

```
train_tagged_words[:5]
```

Out[7]:
```
[('Drink', 'NOUN'),
 ('Carrier', 'NOUN'),
 ('Competes', 'VERB'),
 ('With', 'ADP'),
 ('Cartons', 'NOUN')]
```

In [8]:
```python
tags = {tag for word,tag in train_tagged_words}
print(len(tags))
print(tags)
```

```
12
{'ADJ', 'NUM', 'PRON', 'ADP', 'CONJ', 'VERB', 'NOUN', 'DET', 'PRT', 'X', 'ADV', '.'}
```

In [9]:
```python
vocab = {word for word,tag in train_tagged_words}
```

In [10]:
```python
#compute emission probability
def word_given_tag(word, tag, train_bag = train_tagged_words):
    tag_list = [pair for pair in train_bag if pair[1]==tag]
    count_tag = len(tag_list)
    w_given_tag_list = [pair[0] for pair in tag_list if pair[0]==word]
    count_w_given_tag = len(w_given_tag_list)

    return (count_w_given_tag, count_tag)
```

In [11]:
```python
# compute transition probability
def t2_given_t1(t2, t1, train_bag = train_tagged_words):
    tags = [pair[1] for pair in train_bag]
    count_t1 = len([t for t in tags if t==t1])
    count_t2_t1 = 0
    for index in range(len(tags)-1):
        if tags[index]==t1 and tags[index+1]== t2:
            count_t2_t1 += 1
    return (count_t2_t1, count_t1)
```

In [12]:
```python
tags_matrix = np.zeros((len(tags), len(tags)), dtype='float32')
for i, t1 in enumerate(list(tags)):
    for j, t2 in enumerate(list(tags)):
        tags_matrix[i, j] = t2_given_t1(t2, t1)[0]/t2_given_t1(t2, t1)[1]

print(tags_matrix)
```

```
[[6.33009672e-02 2.17475723e-02 1.94174761e-04 8.05825219e-02
  1.68932043e-02 1.14563107e-02 6.96893215e-01 5.24271838e-03
  1.14563107e-02 2.09708735e-02 5.24271838e-03 6.60194159e-02]
 [3.53445187e-02 1.84219927e-01 1.42806140e-03 3.74866128e-02
  1.42806144e-02 2.07068902e-02 3.51660132e-01 3.57015361e-03
  2.60621198e-02 2.02427700e-01 3.57015361e-03 1.19243130e-01]
 [7.06150308e-02 6.83371304e-03 6.83371304e-03 2.23234631e-02
  5.01138950e-03 4.84738052e-01 2.12756261e-01 9.56719834e-03
  1.41230067e-02 8.83826911e-02 3.69020514e-02 4.19134386e-02]
 [1.07061505e-01 6.32751212e-02 6.96026310e-02 1.69577319e-02
  1.01240189e-03 8.47886596e-03 3.23588967e-01 3.20931405e-01
  1.26550242e-03 3.45482156e-02 1.45532778e-02 3.87243740e-02]
 [1.13611415e-01 4.06147093e-02 6.03732169e-02 5.59824370e-02
  5.48847427e-04 1.50384188e-01 3.49066973e-01 1.23490669e-01
  4.39077942e-03 9.33040585e-03 5.70801310e-02 3.51262353e-02]
 [6.63904250e-02 2.28360966e-02 3.55432779e-02 9.23572779e-02
  5.43278083e-03 1.67955801e-01 1.10589318e-01 1.33609578e-01
  3.06629837e-02 2.15930015e-01 8.38858187e-02 3.48066315e-02]
 [1.25838192e-02 9.14395228e-03 4.65906132e-03 1.76826611e-01
  4.24540639e-02 1.49133503e-01 2.62344331e-01 1.31063312e-02
  4.39345129e-02 2.88252197e-02 1.68945398e-02 2.40094051e-01]
 [2.06410810e-01 2.28546783e-02 3.30602261e-03 9.91806854e-03
  4.31220367e-04 4.02472317e-02 6.35906279e-01 6.03708485e-03
  2.87480245e-04 4.51343954e-02 1.20741697e-02 1.73925534e-02]
 [8.29745606e-02 5.67514673e-02 1.76125243e-02 1.95694715e-02
  2.34833662e-03 4.01174158e-01 2.50489235e-01 1.01369865e-01
  1.17416831e-03 1.21330721e-02 9.39334650e-03 4.50097844e-02]
 [1.76821072e-02 3.07514891e-03 5.41995019e-02 1.42225638e-01
  1.03786280e-02 2.06419379e-01 6.16951771e-02 5.68902567e-02
  1.85085520e-01 7.57255405e-02 2.57543717e-02 1.60868734e-01]
 [1.30721495e-01 2.98681147e-02 1.20248254e-02 1.19472459e-01
  6.98215654e-03 3.39022487e-01 3.21955010e-02 7.13731572e-02
  1.47401085e-02 2.28859577e-02 8.14584941e-02 1.39255241e-01]
 [4.61323895e-02 7.82104954e-02 6.87694475e-02 9.29084867e-02
  6.00793920e-02 8.96899477e-02 2.18538776e-01 1.72191828e-01
  2.78940029e-03 2.56410260e-02 5.25694676e-02 9.23720598e-02]]
```

In [13]:
```
tags_df = pd.DataFrame(tags_matrix, columns = list(tags), index=list(tags))
display(tags_df)
```

	ADJ	NUM	PRON	ADP	CONJ	VERB	NOUN	DET	PRT	X	ADV	.
ADJ	0.063301	0.021748	0.000194	0.080583	0.016893	0.011456	0.696893	0.005243	0.011456	0.020971	0.005243	0.066019
NUM	0.035345	0.184220	0.001428	0.037487	0.014281	0.020707	0.351660	0.003570	0.026062	0.202428	0.003570	0.119243
PRON	0.070615	0.006834	0.006834	0.022323	0.005011	0.484738	0.212756	0.009567	0.014123	0.088383	0.036902	0.041913
ADP	0.107062	0.063275	0.069603	0.016958	0.001012	0.008479	0.323589	0.320931	0.001266	0.034548	0.014553	0.038724
CONJ	0.113611	0.040615	0.060373	0.055982	0.000549	0.150384	0.349067	0.123491	0.004391	0.009330	0.057080	0.035126
VERB	0.066390	0.022836	0.035543	0.092357	0.005433	0.167956	0.110589	0.133610	0.030663	0.215930	0.083886	0.034807
NOUN	0.012584	0.009144	0.004659	0.176827	0.042454	0.149134	0.262344	0.013106	0.043935	0.028825	0.016895	0.240094
DET	0.206411	0.022855	0.003306	0.009918	0.000431	0.040247	0.635906	0.006037	0.000287	0.045134	0.012074	0.017393
PRT	0.082975	0.056751	0.017613	0.019569	0.002348	0.401174	0.250489	0.101370	0.001174	0.012133	0.009393	0.045010
X	0.017682	0.003075	0.054200	0.142226	0.010379	0.206419	0.061695	0.056890	0.185086	0.075726	0.025754	0.160869
ADV	0.130721	0.029868	0.012025	0.119472	0.006982	0.339022	0.032196	0.071373	0.014740	0.022886	0.081458	0.139255
.	0.046132	0.078210	0.068769	0.092908	0.060079	0.089690	0.218539	0.172192	0.002789	0.025641	0.052569	0.092372

In [14]:
```
def Viterbi(words, train_bag = train_tagged_words):
    state = []
    T = list(set([pair[1] for pair in train_bag]))

    for key, word in enumerate(words):
        p = []
        for tag in T:
            if key == 0:
                transition_p = tags_df.loc['.', tag]
            else:
                transition_p = tags_df.loc[state[-1], tag]

            # compute emission and  probability states
            emission_p = word_given_tag(words[key], tag)[0]/word_given_tag(words[key], tag)[1]
            state_probability = emission_p * transition_p
            p.append(state_probability)

        pmax = max(p)

        state_max = T[p.index(pmax)]
        state.append(state_max)
    return list(zip(words, state))
```

In [15]:
```
random.seed(42)
rndom = [random.randint(1,len(test_set)) for x in range(10)]
test_run = [test_set[i] for i in rndom]
test_run_base = [tup for sent in test_run for tup in sent]
test_tagged_words = [tup[0] for sent in test_run for tup in sent]
```

In [16]:
```
start = time.time()
tagged_seq = Viterbi(test_tagged_words)
end = time.time()
difference = end-start
print("Time taken in seconds: ", difference)

check = [i for i, j in zip(tagged_seq, test_run_base) if i == j]
accuracy = len(check)/len(tagged_seq)
print('Viterbi Algorithm Accuracy: ',accuracy*100)
```

```
Time taken in seconds:  18.05593180656433
Viterbi Algorithm Accuracy:  94.94949494949495
```

In [17]:
```
#Takes too long to run
'''
test_tagged_words = [tup for sent in test_set for tup in sent]
test_untagged_words = [tup[0] for sent in test_set for tup in sent]
test_untagged_words

start = time.time()
tagged_seq = Viterbi(test_untagged_words)
end = time.time()
difference = end-start

print("Time taken in seconds: ", difference)

check = [i for i, j in zip(test_tagged_words, test_untagged_words) if i == j]

accuracy = len(check)/len(tagged_seq)
print('Viterbi Algorithm Accuracy: ',accuracy*100)
'''
```

Out[17]:
```
'\ntest_tagged_words = [tup for sent in test_set for tup in sent]\ntest_untagged_words = [tup[0] for sent in test_set
for tup in sent]\ntest_untagged_words\n\nstart = time.time()\ntagged_seq = Viterbi(test_untagged_words)\nend = time.ti
me()\ndifference = end-start\n \nprint("Time taken in seconds: ", difference)\n\ncheck = [i for i, j in zip(test_tagge
d_words, test_untagged_words) if i == j] \n \naccuracy = len(check)/len(tagged_seq)\nprint(\'Viterbi Algorithm Accurac
y: \',accuracy*100)\n'
```

In [18]:
```
patterns = [
    (r'.*ing$', 'VERB'),              # gerund
    (r'.*ed$', 'VERB'),               # past tense
```

```python
    (r'.*es$', 'VERB'),              # verb
    (r'.*\'s$', 'NOUN'),             # possessive nouns
    (r'.*s$', 'NOUN'),               # plural nouns
    (r'\*T?\*?-[0-9]+$', 'X'),       # X
    (r'^-?[0-9]+(.[0-9]+)?$', 'NUM'), # cardinal numbers
    (r'.*', 'NOUN')                  # nouns
]

rule_based_tagger = nltk.RegexpTagger(patterns)
```

In [19]:
```python
def Viterbi_rule_based(words, train_bag = train_tagged_words):
    state = []
    T = list(set([pair[1] for pair in train_bag]))

    for key, word in enumerate(words):
        p = []
        for tag in T:
            if key == 0:
                transition_p = tags_df.loc['.', tag]
            else:
                transition_p = tags_df.loc[state[-1], tag]

            # compute emission and  probability states
            emission_p = word_given_tag(words[key], tag)[0]/word_given_tag(words[key], tag)[1]
            state_probability = emission_p * transition_p
            p.append(state_probability)

        pmax = max(p)
        state_max = rule_based_tagger.tag([word])[0][1]


        if(pmax==0):
            state_max = rule_based_tagger.tag([word])[0][1]
        else:
            if state_max != 'X':
                state_max = T[p.index(pmax)]


        state.append(state_max)
    return list(zip(words, state))
```

In [20]:
```python
start = time.time()
tagged_seq = Viterbi_rule_based(test_tagged_words)
end = time.time()
difference = end-start

print("Time taken in seconds: ", difference)
```

```python
check = [i for i, j in zip(tagged_seq, test_run_base) if i == j]

accuracy = len(check)/len(tagged_seq)
print('Viterbi Algorithm Accuracy: ',accuracy*100)
```

```
Time taken in seconds:  19.284300088882446
Viterbi Algorithm Accuracy:  98.48484848484848
```

In [21]:
```python
test_sent="Will can see Marry"
pred_tags_rule=Viterbi_rule_based(test_sent.split())
pred_tags_withoutRules= Viterbi(test_sent.split())
print(pred_tags_rule)
print(pred_tags_withoutRules)
```

```
[('Will', 'NOUN'), ('can', 'VERB'), ('see', 'VERB'), ('Marry', 'NOUN')]
[('Will', 'ADJ'), ('can', 'VERB'), ('see', 'VERB'), ('Marry', 'ADJ')]
```

Exercise 2:

- Find a new text dataset
- Convert it into csv format
- Redo the same exercise

a & b)

In [23]:
```python
import nltk.corpus
```

In [24]:
```python
nltk.download('brown')
nltk.download('universal_tagset')
```

```
[nltk_data] Downloading package brown to
[nltk_data]     /Users/danielmehta/nltk_data...
[nltk_data]   Package brown is already up-to-date!
[nltk_data] Downloading package universal_tagset to
[nltk_data]     /Users/danielmehta/nltk_data...
[nltk_data]   Package universal_tagset is already up-to-date!
```

Out[24]: True

In [25]:
```python
nltk.corpus.brown.tagged_sents(tagset='universal')
```

Out[25]:
```
[[('The', 'DET'), ('Fulton', 'NOUN'), ('County', 'NOUN'), ('Grand', 'ADJ'), ('Jury', 'NOUN'), ('said', 'VERB'), ('Frid
ay', 'NOUN'), ('an', 'DET'), ('investigation', 'NOUN'), ('of', 'ADP'), ("Atlanta's", 'NOUN'), ('recent', 'ADJ'), ('pri
mary', 'NOUN'), ('election', 'NOUN'), ('produced', 'VERB'), ('``', '.'), ('no', 'DET'), ('evidence', 'NOUN'), ("''",
'.'), ('that', 'ADP'), ('any', 'DET'), ('irregularities', 'NOUN'), ('took', 'VERB'), ('place', 'NOUN'), ('.', '.')],
[('The', 'DET'), ('jury', 'NOUN'), ('further', 'ADV'), ('said', 'VERB'), ('in', 'ADP'), ('term-end', 'NOUN'), ('presen
tments', 'NOUN'), ('that', 'ADP'), ('the', 'DET'), ('City', 'NOUN'), ('Executive', 'ADJ'), ('Committee', 'NOUN'),
(',', '.'), ('which', 'DET'), ('had', 'VERB'), ('over-all', 'ADJ'), ('charge', 'NOUN'), ('of', 'ADP'), ('the', 'DET'),
('election', 'NOUN'), (',', '.'), ('``', '.'), ('deserves', 'VERB'), ('the', 'DET'), ('praise', 'NOUN'), ('and', 'CON
J'), ('thanks', 'NOUN'), ('of', 'ADP'), ('the', 'DET'), ('City', 'NOUN'), ('of', 'ADP'), ('Atlanta', 'NOUN'), ("''",
'.'), ('for', 'ADP'), ('the', 'DET'), ('manner', 'NOUN'), ('in', 'ADP'), ('which', 'DET'), ('the', 'DET'), ('electio
n', 'NOUN'), ('was', 'VERB'), ('conducted', 'VERB'), ('.', '.')], ...]
```

In [26]:
```python
brown_tagged = nltk.corpus.brown.tagged_sents(tagset='universal')[:500]
```

In [27]:
```python
flattened = [(word, tag) for sent in brown_tagged for (word, tag) in sent]
```

In [28]:
```python
df = pd.DataFrame(flattened, columns=['Word', 'Tag'])
df.to_csv('brown_pos.csv', index=False)
```

c)

In [30]:
```python
#Split training and validation 80/20
train_sents, test_sents = train_test_split(brown_tagged, train_size=0.8, random_state=5501)
```

In [31]:
```python
train_tagged_words = [tup for sent in train_sents for tup in sent]
test_tagged_words = [tup for sent in test_sents for tup in sent]
```

In [32]:
```python
print("Train:", len(train_tagged_words))
print("Test:", len(test_tagged_words))
```

```
Train: 9492
Test: 2219
```

In [33]:
```python
def word_given_tag(word, tag, train_bag = train_tagged_words):
    tag_list = [pair for pair in train_bag if pair[1]==tag]
    count_tag = len(tag_list)
    w_given_tag_list = [pair[0] for pair in tag_list if pair[0]==word]
    count_w_given_tag = len(w_given_tag_list)

    return (count_w_given_tag, count_tag)
```

In [34]:
```python
def t2_given_t1(t2, t1, train_bag = train_tagged_words):
    tags = [pair[1] for pair in train_bag]
    count_t1 = len([t for t in tags if t==t1])
    count_t2_t1 = 0
```

```python
    for index in range(len(tags)-1):
        if tags[index]==t1 and tags[index+1]== t2:
            count_t2_t1 += 1
    return (count_t2_t1, count_t1)
```

In [35]:
```python
tags = sorted(list({tag for _, tag in train_tagged_words}))
```

In [36]:
```python
tags_matrix = np.zeros((len(tags), len(tags)), dtype='float32')
for i, t1 in enumerate(list(tags)):
    for j, t2 in enumerate(list(tags)):
        tags_matrix[i, j] = t2_given_t1(t2, t1)[0]/t2_given_t1(t2, t1)[1]

print(tags_matrix)
```

```
[[0.10965795 0.04426559 0.09959759 0.04627766 0.05633803 0.18812877
  0.23138833 0.02615694 0.07645875 0.02012073 0.10060363 0.        ]
 [0.05322581 0.06129032 0.0483871  0.00322581 0.02741935 0.00322581
  0.75       0.01129032 0.         0.02580645 0.01451613 0.0016129 ]
 [0.01071723 0.08244023 0.02555647 0.00741962 0.0016488  0.4286892
  0.30008245 0.05441055 0.03050289 0.00824402 0.05028854 0.        ]
 [0.08       0.12       0.15272728 0.06909091 0.01454545 0.06909091
  0.02181818 0.00727273 0.04       0.04       0.38545454 0.        ]
 [0.00921659 0.14285715 0.05990783 0.05529954 0.         0.16129032
  0.33179724 0.01382488 0.04147466 0.02304148 0.16129032 0.        ]
 [0.01358696 0.21014492 0.01086957 0.00905797 0.         0.00543478
  0.6512681  0.02173913 0.00724638 0.00271739 0.06793478 0.        ]
 [0.22210708 0.017962   0.23039724 0.01692573 0.03937824 0.01761658
  0.25423142 0.00725389 0.01139896 0.02901554 0.1537133  0.        ]
 [0.14857143 0.10285714 0.14857143 0.00571429 0.04       0.02857143
  0.41714287 0.06285714 0.01714286 0.00571429 0.02285714 0.        ]
 [0.06493507 0.004329   0.05194805 0.04761905 0.00865801 0.02164502
  0.         0.         0.01298701 0.01298701 0.7748918  0.        ]
 [0.03846154 0.00854701 0.06837607 0.02136752 0.0042735  0.04273504
  0.03846154 0.         0.         0.0042735  0.77350426 0.        ]
 [0.06984334 0.04503917 0.17297651 0.07245431 0.00913838 0.17167102
  0.14360313 0.00979112 0.03328982 0.05221932 0.21997389 0.        ]
 [0.         0.         0.         0.         0.         0.
  0.5        0.         0.         0.         0.         0.5       ]]
```

In [37]:
```python
tags_matrix = np.zeros((len(tags), len(tags)), dtype='float32')

for i, t1 in enumerate(tags):
    for j, t2 in enumerate(tags):
        count_t2_t1, count_t1 = t2_given_t1(t2, t1)
        if count_t1 == 0:
            tags_matrix[i, j] = 0.0
        else:
```

```
            tags_matrix[i, j] = count_t2_t1 / count_t1

tags_df = pd.DataFrame(tags_matrix, columns=tags, index=tags)
print(tags_df)
```

```
             .       ADJ       ADP       ADV      CONJ       DET      NOUN  \
.     0.109658  0.044266  0.099598  0.046278  0.056338  0.188129  0.231388
ADJ   0.053226  0.061290  0.048387  0.003226  0.027419  0.003226  0.750000
ADP   0.010717  0.082440  0.025556  0.007420  0.001649  0.428689  0.300082
ADV   0.080000  0.120000  0.152727  0.069091  0.014545  0.069091  0.021818
CONJ  0.009217  0.142857  0.059908  0.055300  0.000000  0.161290  0.331797
DET   0.013587  0.210145  0.010870  0.009058  0.000000  0.005435  0.651268
NOUN  0.222107  0.017962  0.230397  0.016926  0.039378  0.017617  0.254231
NUM   0.148571  0.102857  0.148571  0.005714  0.040000  0.028571  0.417143
PRON  0.064935  0.004329  0.051948  0.047619  0.008658  0.021645  0.000000
PRT   0.038462  0.008547  0.068376  0.021368  0.004274  0.042735  0.038462
VERB  0.069843  0.045039  0.172977  0.072454  0.009138  0.171671  0.143603
X     0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.500000

           NUM      PRON       PRT      VERB         X
.     0.026157  0.076459  0.020121  0.100604  0.000000
ADJ   0.011290  0.000000  0.025806  0.014516  0.001613
ADP   0.054411  0.030503  0.008244  0.050289  0.000000
ADV   0.007273  0.040000  0.040000  0.385455  0.000000
CONJ  0.013825  0.041475  0.023041  0.161290  0.000000
DET   0.021739  0.007246  0.002717  0.067935  0.000000
NOUN  0.007254  0.011399  0.029016  0.153713  0.000000
NUM   0.062857  0.017143  0.005714  0.022857  0.000000
PRON  0.000000  0.012987  0.012987  0.774892  0.000000
PRT   0.000000  0.000000  0.004274  0.773504  0.000000
VERB  0.009791  0.033290  0.052219  0.219974  0.000000
X     0.000000  0.000000  0.000000  0.000000  0.500000
```

In [38]:

```
def Viterbi(words, train_bag=train_tagged_words):
    state = []
    T = list(set([tag for _, tag in train_bag]))

    for i, word in enumerate(words):
        p = []
        for tag in T:
            if i == 0:
                trans_p = tags_df.loc['.',tag] if '.' in tags_df.index else 1e-6
            else:
                trans_p = tags_df.loc[state[-1],tag] if state[-1] in tags_df.index else 1e-6

            # Emission
            emission_count, tag_count = word_given_tag(word, tag)
            emission_p = emission_count /tag_count if tag_count > 0 else 1e-6
```

```python
            # Combined prob
            state_p = emission_p * trans_p
            p.append(state_p)

        max_p = max(p)
        max_state = T[p.index(max_p)]
        state.append(max_state)

    return list(zip(words, state))
```

In [39]:
```python
start = time.time()

test_words = [word for word, _ in test_tagged_words]
tagged_seq = Viterbi(test_words)

end = time.time()
difference = end - start

print("Time taken in seconds:", round(difference, 4))
correct = [pred for pred, actual in zip(tagged_seq, test_tagged_words) if pred == actual]
accuracy = len(correct) / len(tagged_seq)

print("Viterbi Algorithm Accuracy:", round(accuracy * 100, 2), "%")
```

```
Time taken in seconds: 12.2067
Viterbi Algorithm Accuracy: 82.47 %
```

Exercise 3: Markov Chains in Python with Model Examples

In [41]:
```python
import numpy as np
import random as rm
```

In [42]:
```python
states = ["Sleep","Icecream","Run"]

transitionName = [["SS","SR","SI"],["RS","RR","RI"],["IS","IR","II"]]

transitionMatrix = [[0.2,0.6,0.2],[0.1,0.6,0.3],[0.2,0.7,0.1]]
```

In [43]:
```python
if sum(transitionMatrix[0])+sum(transitionMatrix[1])+sum(transitionMatrix[1]) != 3:
    print("Somewhere, something went wrong. Transition matrix, perhaps?")
else: print("All is gonna be okay, you should move on!! ;)")
```

```
All is gonna be okay, you should move on!! ;)
```

In [44]:

```python
# A function that implements the Markov model to forecast the state/mood.
def activity_forecast(days):
    # Choose the starting state
    activityToday = "Sleep"
    print("Start state: " + activityToday)
    # Shall store the sequence of states taken. So, this only has the starting state for now.
    activityList = [activityToday]
    i = 0
    # To calculate the probability of the activityList
    prob = 1
    while i != days:
        if activityToday == "Sleep":
            change = np.random.choice(transitionName[0],replace=True,p=transitionMatrix[0])
            if change == "SS":
                prob = prob * 0.2
                activityList.append("Sleep")
                pass
            elif change == "SR":
                prob = prob * 0.6
                activityToday = "Run"
                activityList.append("Run")
            else:
                prob = prob * 0.2
                activityToday = "Icecream"
                activityList.append("Icecream")
        elif activityToday == "Run":
            change = np.random.choice(transitionName[1],replace=True,p=transitionMatrix[1])
            if change == "RR":
                prob = prob * 0.5
                activityList.append("Run")
                pass
            elif change == "RS":
                prob = prob * 0.2
                activityToday = "Sleep"
                activityList.append("Sleep")
            else:
                prob = prob * 0.3
                activityToday = "Icecream"
                activityList.append("Icecream")
        elif activityToday == "Icecream":
            change = np.random.choice(transitionName[2],replace=True,p=transitionMatrix[2])
            if change == "II":
                prob = prob * 0.1
                activityList.append("Icecream")
                pass
            elif change == "IS":
                prob = prob * 0.2
```

```
                activityToday = "Sleep"
                activityList.append("Sleep")
            else:
                prob = prob * 0.7
                activityToday = "Run"
                activityList.append("Run")
        i += 1
    print("Possible states: " + str(activityList))
    print("End state after "+ str(days) + " days: " + activityToday)
    print("Probability of the possible sequence of states: " + str(prob))

# Function that forecasts the possible state for the next 2 days
activity_forecast(2)
```

```
Start state: Sleep
Possible states: ['Sleep', 'Sleep', 'Run']
End state after 2 days: Run
Probability of the possible sequence of states: 0.12
```

In [45]:

```
def activity_forecast(days):
    # Choose the starting state
    activityToday = "Sleep"
    activityList = [activityToday]
    i = 0
    prob = 1
    while i != days:
        if activityToday == "Sleep":
            change = np.random.choice(transitionName[0],replace=True,p=transitionMatrix[0])
            if change == "SS":
                prob = prob * 0.2
                activityList.append("Sleep")
                pass
            elif change == "SR":
                prob = prob * 0.6
                activityToday = "Run"
                activityList.append("Run")
            else:
                prob = prob * 0.2
                activityToday = "Icecream"
                activityList.append("Icecream")
        elif activityToday == "Run":
            change = np.random.choice(transitionName[1],replace=True,p=transitionMatrix[1])
            if change == "RR":
                prob = prob * 0.5
                activityList.append("Run")
                pass
            elif change == "RS":
                prob = prob * 0.2
```

```python
                activityToday = "Sleep"
                activityList.append("Sleep")
            else:
                prob = prob * 0.3
                activityToday = "Icecream"
                activityList.append("Icecream")
        elif activityToday == "Icecream":
            change = np.random.choice(transitionName[2],replace=True,p=transitionMatrix[2])
            if change == "II":
                prob = prob * 0.1
                activityList.append("Icecream")
                pass
            elif change == "IS":
                prob = prob * 0.2
                activityToday = "Sleep"
                activityList.append("Sleep")
            else:
                prob = prob * 0.7
                activityToday = "Run"
                activityList.append("Run")
        i += 1
    return activityList

# To save every activityList
list_activity = []
count = 0

# `Range` starts from the first count up until but excluding the last count
for iterations in range(1,10000):
        list_activity.append(activity_forecast(2))

# Check out all the `activityList` we collected
#print(list_activity)

# Iterate through the list to get a count of all activities ending in state:'Run'
for smaller_list in list_activity:
    if(smaller_list[2] == "Run"):
        count += 1

# Calculate the probability of starting from state:'Sleep' and ending at state:'Run'
percentage = (count/10000) * 100
print("The probability of starting at state:'Sleep' and ending at state:'Run'= " + str(percentage) + "%")
```

```
The probability of starting at state:'Sleep' and ending at state:'Run'= 61.95%
```

```
In [ ]:
```