DS8004 Project 2 – Hidden Markov Models

Author: Rafik Matta

Contents

Project Description ... 3
Solution ... 3
Overview ... 3
Definition of Hidden Markov Model (HMM) and Key equations ... 3
Baum-Welch Algorithm ... 3
Viterbi Algorithm ... 4
Experiment Design ... 5
Results ... 6
Exploratory Data Analysis ... 6
Statistics of Variables ... 7
Analysis ... 7
Conclusions and Findings ... 8
References ... 8
Appendix ... 9
Baum-Welch Algorithm Implementation ... 9
Normalized Forward Algorithm Normalization Calculation ... 9
Normalized Forward Algorithm ... 9
Normalized Backward Algorithm ... 9
Gamma Calculation ... 9
Xi Calculation ... 9
Baum Welch Implementation ... 10
Viterbi Algorithm ... 11
Exploratory Data Analysis ... 11
Discretizing Attributes ... 11
Analysis ... 11
CO2 ... 11
Light Level ... 12

Project Description

In this project we are modeling a learning problem as a Hidden Markov Model and applying the Baum Welch (BW) Algorithm to learn transition probabilities, emission probabilities and initial state probabilities. We are then using the Viterbi Algorithm to predict the state sequence based on the learned parameters and observations.

Solution

Overview

Two algorithms were implemented. Both the Baum Welch and Viterbi algorithms were implemented based on the description and equations found in the Introduction to Machine Learning 3e text book by Etham Alpaydin and a key paper describing practical considerations of the implementation. Two attributes were selected to serve as observations sequences based on the results of the EDA.

Definition of Hidden Markov Model (HMM) and Key equations

The definition of an HMM that is being used is a system with S = $\{s_{1,}...,s_N\}$ states, where N is the number of states and V = $\{v_{1,}...,v_M\}$ where M is the set of possible symbols (alphabet) in our vocabulary. The transition probability matrix is defined as A = $\{a_{ij}\}$ and emission probability matrix as B = $\{b_i(v_k)\}$. Finally, the initial state probabilities are, Π = $\{\Pi_i\}$.

The HMM model parameters are denoted as follows, $\lambda = (\Pi, A, B)$. (Zhai, 2003)

The following constraints apply to the parameters (Zhai, 2003):

$$\sum_{i=1}^{N} \pi_i = 1$$
$$\sum_{j=1}^{N} a_{ij} = 1 \text{ for } i = 1, 2, ..., N$$
$$\sum_{k=1}^{M} b_i(v_k) = 1 \text{ for } i = 1, 2, ..., N$$

Baum-Welch Algorithm

The BW algorithm (Zhai, 2003) is applied to the data to predict the model parameters from an initialized set. The practical implementation of the algorithm slightly differs from the one found in the Alpaydin Text. Since the time horizon of the dataset is quite long, the Forward-Backward algorithm suffers from underflow. To overcome this, normalization is applied to the both the Forward and Backward algorithm as follows:

<u>Forward Algorithm </u>(Zhai, 2003)

1. $\hat{\alpha}_1(i) = \frac{\pi_i b_i(o_1)}{\sum_{k=1}^{N} \pi_k b_i(o_1)}$

2. For $1 \le i < T$, $\hat{\alpha}_{t+1}(i) = \frac{b_i(o_{t+1}) \sum_{j=1}^{N} \hat{\alpha}_t(j) a_{ji}}{\sum_{k=1}^{N} b_k(o_{t+1}) \sum_{j=1}^{N} \hat{\alpha}_t(j) a_{jk}}$

<u>Backward Algorithm </u>(Zhai, 2003)

1. $\hat{\beta}_T(i) = \beta_T(i) = 1$
2. For $1 \le t < T$, $\hat{\beta}_t(i) = \eta_{t+1} \sum_{j=1}^{N} \hat{\beta}_{t+1}(j) a_{ij} b_j(o_{t+1})$

For the BW algorithm, we calculate γ which is the probability of being at state s_i at time t given our observations (Alpaydin, 2014):

$$\begin{aligned}\gamma_t(i) &\equiv P(q_t = S_i \mid O, \lambda) \\ &= \frac{\alpha_t(i)\beta_t(i)}{\sum_{j=1}^{N} \alpha_t(j)\beta_t(j)}\end{aligned}$$

Further, we calculate ξ, which is the probability of going through a transition from state i to j at time t given the observations (Alpaydin, 2014):

$$\xi_t(i,j) \equiv P(q_t = S_i, q_{t+1} = S_j \mid O, \lambda)$$

$$\xi_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{\sum_k \sum_l \alpha_t(k) a_{kl} b_l(O_{t+1}) \beta_{t+1}(l)}$$

No normalization needs to be applied to either of the above terms, as the normalization factor is canceled out. Finally, as we do a full pass through an observation sequence, the initial model parameters are updated as such (Alpaydin, 2014):

$$\pi'_i = \gamma_1(i)$$

$$a'_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{j=1}^{N} \sum_{t=1}^{T-1} \xi_t(i,j)}$$

$$b_i(v_k)' = \frac{\sum_{t=1, o_t = v_k}^{T} \gamma_t(i)}{\sum_{t=1}^{T} \gamma_t(i)}$$

These new parameters are fed back into the algorithm until convergence.

For briefness, further detail on the implementation will be omitted and the references provided can be further explored.

Viterbi Algorithm

The Viterbi Algorithm is implemented as described in the Alpaydin text (Alpaydin, 2014).

The initialization step is:

$\delta_1(i) = \pi_i b_i(O_1)$, $\psi_1(i) = 0$

The recursive step is as follows:

$\delta_t(j) = \max_i \delta_{t-1}(i) a_{ij} b_j(O_t)$, $\psi_t(j) = \text{argmax}_i\ \delta_{t-1}(i) a_{ij}$

Finally, at termination we determine the T'th state using the following:

$$p^* = \max_i \delta_T(i),\ q_T^* = \operatorname{argmax}_i \delta_T(i)$$

Once this is done, we can use dynamic programming to backtrack and determine the path given the state at final time T.

$$q_t^* = \psi_{t+1}(q_{t+1}^*),\ t=T-1, T-2, \ldots, 1$$

Experiment Design

The following steps were taken to run the experiment:

1. Randomly generate transition matrix and initial state vector (pie). Emission probabilities are determined through exploratory data analysis for both attributes.
2. Apply Baum Welch until convergence (this was empirically determined to be roughly 15-20 iterations)
3. Apply Viterbi algorithm to predict state sequence.

Results

Exploratory Data Analysis

Dataset: https://archive.ics.uci.edu/ml/datasets/Occupancy+Detection

Within the dataset, there are a total of 6 attributes. For the project, ‘CO2’ was assigned and based on the results of the exploratory analysis, both light and temperature were explored, but it was determined based on visual analysis of the plots of each attribute, that ‘Light’ had a closer correlation to ‘Occupancy’. Both ‘CO2’ and ‘Light’ were discretized into ‘CO2_Level’ and ‘Light_Level’. Figure 1 shows the plots of each one of the attributes against time.

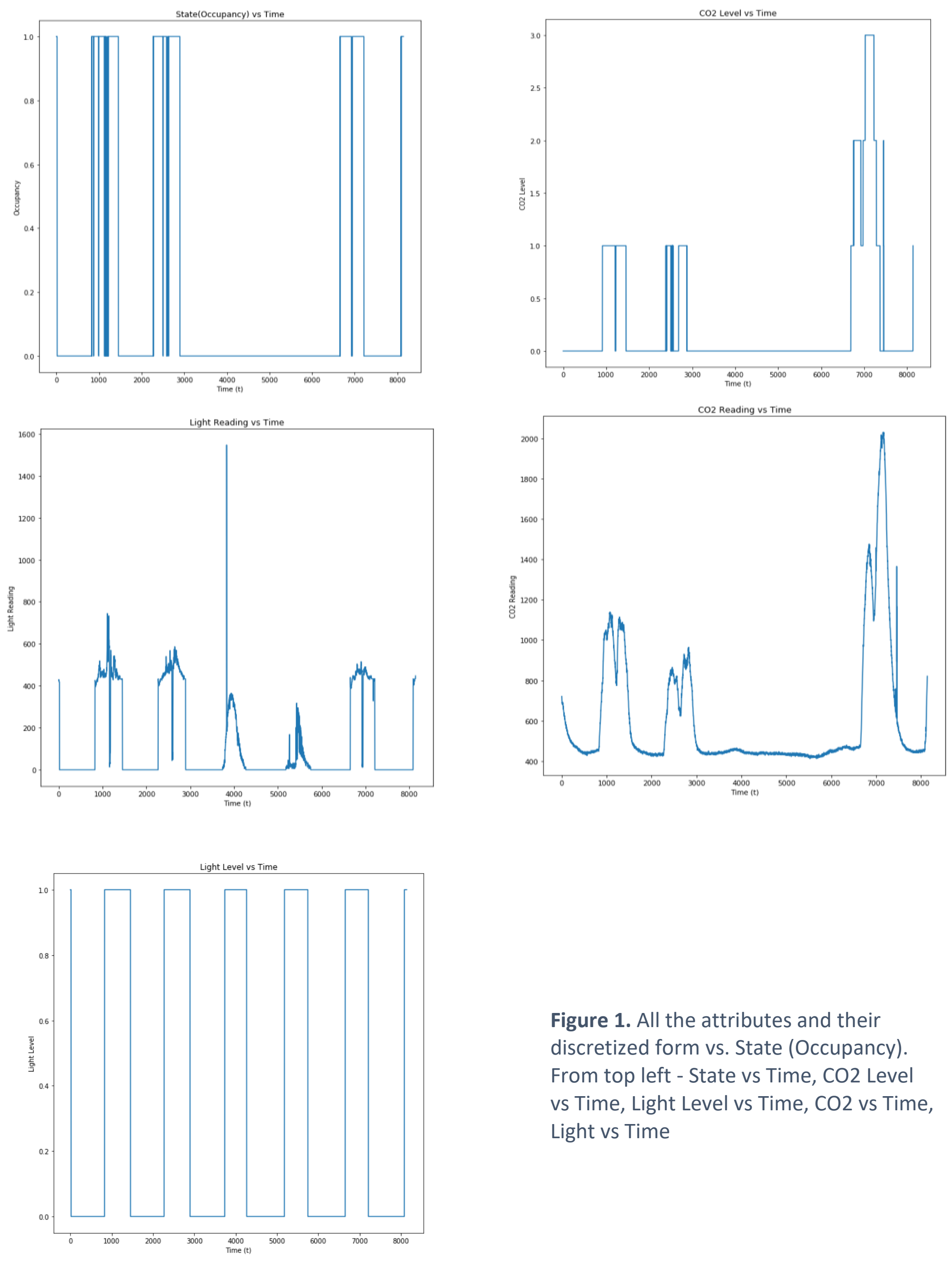
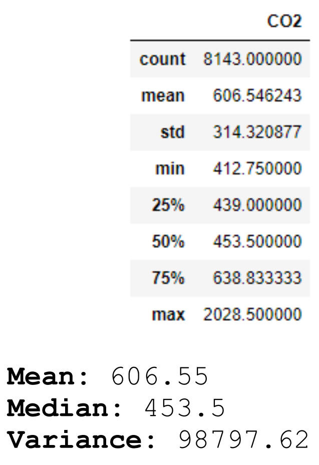


Figure 1. All the attributes and their discretized form vs. State (Occupancy). From top left - State vs Time, CO2 Level vs Time, Light Level vs Time, CO2 vs Time, Light vs Time

As can be seen in the above plots, ‘Light’ and ‘Light_Level’ are not as well correlated with ‘Occupancy’. This did reflect in the performance of the prediction as can be seen in the Results section.

Statistics of Variables

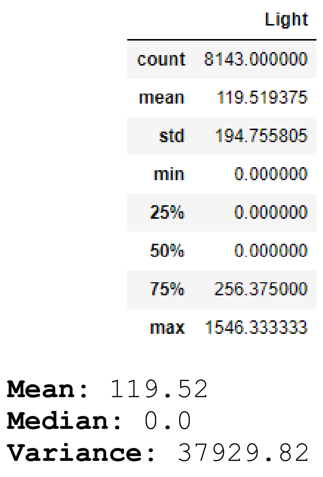
Below are the basic statistics of the variables.



	CO2
count	8143.000000
mean	606.546243
std	314.320877
min	412.750000
25%	439.000000
50%	453.500000
75%	638.833333
max	2028.500000

```
Mean: 606.55
Median: 453.5
Variance: 98797.62
```

Figure 2. CO2 Statistics



	Light
count	8143.000000
mean	119.519375
std	194.755805
min	0.000000
25%	0.000000
50%	0.000000
75%	256.375000
max	1546.333333

```
Mean: 119.52
Median: 0.0
Variance: 37929.82
```

Figure 3. Light Statistics

The variance in the ‘Light’ attribute is much lower than ‘CO2’. This was accounted for in the discretization, where Light really only has two possible emissions (on/off). The attribute in and of itself cannot be a perfect reflection of occupancy as many things can contribute to the lighting in a room (day light). That being said, given the low variance, and the median being 0.0, any value above 0 was considered as an observation of light being in the room and increase the possibility of their being a person in the room, as in all cases if a person is in the room there’s a high likelihood of their being light.

Analysis

An interval of Δt = 100 was chosen as there are a total of 8138 (or minutes) valid ticks and this was too large for the BW and Viterbi algorithms to run efficiently. Therefore, we are checking the observation every 100 minutes. The Analysis section shows the results of the runs.

Both algorithms were applied to each sequence. Since the sequences aren’t necessarily independent, they were not treated as such and the BW algorithm was applied to each sequence and the parameters were learned

ATTRIBUTE	**ITERATIONS**	**ΔT**	**ACCURACY**
CO2_LEVEL	20	100	83%
CO2_LEVEL	20	50	87%
LIGHT_LEVEL	20	100	73%
LIGHT_LEVEL	20	50	79%

Conclusions and Findings

Empirically, the optimal number of iterations was determined to be between 15-20 runs of the BW algorithm before convergence was achieved. Further, the initial probabilities for the transition and emission matrices and initial state probabilities had only a minor effect on the performance of the algorithm and that was more related to the length of time it took to converge. As such this discussion will be more focused on the choice of attribute and the Δt as parameters of concern. It's immediately clear from the results that since the 'Light' attribute didn't correlate as well as the 'CO2' that it's performance in prediction suffered. But the choice of Δt also made quite a difference. As Δt became smaller, the gap closed. This indicates that over a long sequence of time, the differences were not as pronounced between the two observation sequences. Given the overall accuracy of the predictions for both attributes, it's clear that this dataset and problem models well as an HMM. Further improvements can be made to the prediction performance, especially with relation the Δt but that would require a more parallelized implementation of the algorithm to allow training to occur in a reasonable time. As well, as the observation sequence grows, the prediction confidence is lowered. There's also risk of overfitting to the training data as Δt is increased.

References

Alpaydin, E. (2014). Introduction to Machine Learning, 3e.

Zhai, C. (2003). *A Brief Note on the Hidden Markov Models (HMMs).*

Appendix

Below is the code used in the project.

Baum-Welch Algorithm Implementation

Normalized Forward Algorithm Normalization Calculation

```python
def alpha_ct_calc(x,c_t,t,observations,a_matrix,b_matrix,big_t):
    if t==big_t:
        return c_t
    else:
        t = t + 1
        temp1 = np.asarray(x*a_matrix)[0]
        #print(temp1)
        temp2 = np.asarray(b_matrix[observations[t],:])[0]
        #print(temp2)
        x= temp1*temp2
        c = sum(x)
        c_t.append(c)
        x = x/c
        return alpha_ct_calc(x,c_t,t,observations,a_matrix,b_matrix,big_t)
```

Normalized Forward Algorithm

```python
def alpha_var(x,c_t,t,observations,a_matrix,b_matrix,big_t):
    if t==big_t:
        #print(x)
        return(x)
    else:
        t = t + 1
        temp1 = np.asarray(x*a_matrix)[0]
        temp2 = np.asarray(b_matrix[observations[t],:])[0]
        x= temp1*temp2
        x = x/c_t[t]
        return alpha_var(x,c_t,t,observations,a_matrix,b_matrix,big_t)
```

Normalized Backward Algorithm

```python
def beta_var(x,c_t,t,observations,a_matrix,b_matrix,small_t):
    if t==small_t:
        return(x)
    else:
        t = t - 1
        x_new = np.zeros(len(x))
        for i in range(0,len(x)):
            temp1 = np.asarray(a_matrix[i,:])[0]
            #print(temp1)
            temp2 = np.asarray(b_matrix[observations[t+1],:])[0]
            #print(temp2)
            temp3 = temp1*temp2*x
            #print(temp3)
            x_new[i] = sum(temp3)
        x_new = x_new/c_t[t]
        return beta_var(x_new,c_t,t,observations,a_matrix,b_matrix,small_t)
```

Gamma Calculation

```python
def gamma_vec(alpha,beta):
    gamma = (alpha*beta)/sum(alpha*beta)
    return(gamma)
```

Xi Calculation

```python
def xi_matrix(alpha,beta,a_matrix,b_matrix,observation):
    xi_matrix = np.empty([a_matrix.shape[0],a_matrix.shape[1]])
    denominator = 0

    for k in range(0,a_matrix.shape[0]):
        for l in range(0,a_matrix.shape[1]):
            denominator = denominator + alpha[k]*a_matrix[k,l]*b_matrix[observation,l]*beta[l]

    for i in range(0,a_matrix.shape[0]):
        for j in range(0,a_matrix.shape[0]):
            numerator = alpha[i]*a_matrix[i,j]*b_matrix[observation,j]*beta[j]
            xi_matrix[i,j] = numerator/denominator

    return(xi_matrix)
```

Baum Welch Implementation

```python
def bw_algo(observations,pi_vector,a_matrix,b_matrix,alphabet,iterations):

    for update in range(0,iterations):
        print("==========================================")
        print("Iteration #:" + str(update))

        #generic initializations
        big_t = len(observations)

        gamma = np.zeros([big_t,pi_vector.shape[0]])
        xi = np.zeros([big_t,a_matrix.shape[0],a_matrix.shape[1]])

        c_t = []

        alpha_1 = pi_vector*np.asarray(b_matrix[observations[0],:])
        if(alpha_1.shape[0] == 1):
            alpha_1 = alpha_1[0]
        c = sum(alpha_1)
        alpha_1 = alpha_1/c
        c_t.append(c)

        alpha_ct = []
        alpha_ct = alpha_ct_calc(alpha_1,c_t,0,observations,a_matrix,b_matrix,big_t-1)

        beta_1 = np.ones(pi_vector.shape[0])
        gamma = gamma_vec(alpha_1,beta_1)

        beta_t_plus_1 = np.ones(pi_vector.shape[0])
        beta_t_plus_1 = beta_var(beta_1,alpha_ct,big_t-1,observations,a_matrix,b_matrix,1)

        xi = xi_matrix(alpha_1,beta_t_plus_1,a_matrix,b_matrix,observations[1])

        a_matrix_new = np.zeros([a_matrix.shape[0],a_matrix.shape[1]])

        #for a matrix(transition probabilities matrix)
        for t in range(1,big_t-2):
            alpha_t = alpha_var(alpha_1,alpha_ct,0,observations,a_matrix,b_matrix,t)
            beta_t = beta_var(beta_1,alpha_ct,big_t-1,observations,a_matrix,b_matrix,t-1)
            beta_t_plus_1 = beta_var(beta_1,alpha_ct,big_t-1,observations,a_matrix,b_matrix,t)
            gamma = gamma + gamma_vec(alpha_t,beta_t)
            xi = xi + xi_matrix(alpha_t,beta_t_plus_1,a_matrix,b_matrix,observations[t+1])

        for i in range(0,a_matrix.shape[0]):
            for j in range(0,a_matrix.shape[1]):
                a_matrix_new[i,j] = xi[i,j]/gamma[i]


        #for b matrix(emission prob matrix) updates
        b_matrix_new = np.zeros([b_matrix.shape[0],b_matrix.shape[1]])

        for m in range(0,len(alphabet)):
            gamma = gamma_vec(alpha_1,beta_1)
            #t=0 calculations happen here
            if(observations[t] == alphabet[m]):
                num_vec = gamma
            else:
                num_vec = np.zeros([a_matrix.shape[0]])

            denom_vec = gamma

            #for loop starts from t=1->T
            for t in range(1,big_t-1):
                alpha_t = alpha_var(alpha_1,alpha_ct,0,observations,a_matrix,b_matrix,t)
                beta_t = beta_var(beta_1,alpha_ct,big_t-1,observations,a_matrix,b_matrix,t-1)
                gamma = gamma_vec(alpha_t,beta_t)
                denom_vec = denom_vec + gamma
                if(observations[t] == alphabet[m]):
                    num_vec = num_vec + gamma

            result_vec = num_vec/denom_vec

            for j in range(0,b_matrix.shape[1]):
                b_matrix_new[m,j] = result_vec[j]

        new_pi_vector = gamma_vec(alpha_1,beta_1)

        a_matrix = np.asmatrix(a_matrix_new)
        b_matrix = np.asmatrix(b_matrix_new)
        pi_vector = np.asarray(new_pi_vector)

        print("a_matrix: ")
        print(a_matrix)
        print("b_matrix: ")
        print(b_matrix)
        print("pi_vector: ")
        print(pi_vector)
    return(a_matrix,b_matrix,pi_vector)
```

Viterbi Algorithm

```
def viterbi(v,seq,a_matrix,b_matrix,observations,t,big_t):
    if(t == big_t):
        return(max(v[t]),np.argmax(v[t]))
    else:
        t= t+1
        v_temp = np.zeros(v.shape[1])
        seq_temp = np.zeros(seq.shape[1])

        for j in range(0,a_matrix.shape[1]):
            v_temp[j] = v[t-1,0]*a_matrix[0,j]*b_matrix[observations[t],j]
            seq_temp[j] = 0
            for i in range(0,a_matrix.shape[0]):
                temp_var = (v[t-1,i]*a_matrix[i,j]*b_matrix[observations[t],j])
                if(temp_var > v_temp[j]):
                    v_temp[j] = temp_var
                    seq_temp[j] = i

        #v_temp =v_temp/sum(v_temp)
        v[t] = v_temp
        seq[t] = seq_temp
        print(v[t])
        print(seq[t])
        return viterbi(v,seq,a_matrix,b_matrix,observations,t,big_t)
```

Exploratory Data Analysis

```
data_set_1 = pd.read_csv('datatraining.csv')
```

Discretizing Attributes

```
new_data = []
for data in data_set_1['Light']:
    val = data
    if val == 0:
        new_data.append(0)
    elif val > 0:
        new_data.append(1)

data_set_1['Light_Level'] = pd.DataFrame(new_data)
```

```
new_data = []
for data in data_set_1['CO2']:
    val = data
    if (val > 400.00) & (val < 800.00):
        new_data.append(0)
    elif (val > 800) & (val < 1200):
        new_data.append(1)
    elif (val > 1200) & (val < 1600):
        new_data.append(2)
    elif (val > 1600):
        new_data.append(3)

data_set_1['CO2_Level'] = pd.DataFrame(new_data)
```

Analysis

CO2

```
a_matrix = np.matrix([[0.5,0.5],
                      [0.6,0.4]])
b_matrix = np.matrix([[0.2,0.1],
                      [0.3,0.5],
                      [0.2,0.3],
                      [0.3,0.2]])
pi_vector = np.array([0.2,0.8])

delta_t = 50
alphabet = [0,1,2,3]
observations = data_set_1[pd.notnull(data_set_1['CO2_Level'])]['CO2_Level'].tolist()
observations = [int(x) for x in observations]
observations = observations[0:len(observations):delta_t]

new_a_matrix, new_b_matrix, new_pi = bw_algo(observations,pi_vector,a_matrix,b_matrix,alphabet,20)
```

```
v_matrix = np.zeros([len(observations),len(new_pi)])
seq_matrix = np.zeros([len(observations),len(new_pi)])
v1 = pi_vector*np.asarray(new_b_matrix[observations[0],:])[0]
#v1 = v1/sum(v1)
seq1 = np.zeros(len(new_pi))
v_matrix[0] = v1
seq_matrix[0] = seq1
viterbi(v_matrix,seq_matrix,new_a_matrix,new_b_matrix,observations,0,len(observations)-1)
```

```
predicted_sequence = []
for i in range(0,v_matrix.shape[0]):
    predicted_sequence.append(seq_matrix[i,np.argmax(v_matrix[i])])
```

```
actual_sequence = data_set_1[pd.notnull(data_set_1['Occupancy'])]['Occupancy'].tolist()
actual_sequence = actual_sequence[0:len(actual_sequence):delta_t]
```

```
correct = 0
total = len(actual_sequence)
for i in range(0,len(actual_sequence)):
    difference = actual_sequence[i]-predicted_sequence[i]
    if difference == 0.0:
        correct = correct + 1
print(float(correct)/float(total))
```

Light Level

```
a_matrix = np.matrix([[0.5,0.5],
                      [0.6,0.4]])
b_matrix = np.matrix([[0.7,0.3],
                      [0.3,0.7]])
pi_vector = np.array([0.2,0.8])

delta_t = 50

alphabet = [0,1]
observations = data_set_1[pd.notnull(data_set_1['Light_Level'])]['Light_Level'].tolist()
observations = [int(x) for x in observations]
observations = observations[0:len(observations):delta_t]

new_a_matrix, new_b_matrix, new_pi = bw_algo(observations,pi_vector,a_matrix,b_matrix,alphabet,20)
```

```
v_matrix = np.zeros([len(observations),len(new_pi)])
seq_matrix = np.zeros([len(observations),len(new_pi)])
v1 = pi_vector*np.asarray(new_b_matrix[observations[0],:])[0]
#v1 = v1/sum(v1)
seq1 = np.zeros(len(new_pi))
v_matrix[0] = v1
seq_matrix[0] = seq1
viterbi(v_matrix,seq_matrix,new_a_matrix,new_b_matrix,observations,0,len(observations)-1)
```

```
predicted_sequence = []
for i in range(0,v_matrix.shape[0]):
    predicted_sequence.append(seq_matrix[i,np.argmax(v_matrix[i])])
```

```
actual_sequence = data_set_1[pd.notnull(data_set_1['Occupancy'])]['Occupancy'].tolist()
actual_sequence = actual_sequence[0:len(actual_sequence):delta_t]
```

```
correct = 0
total = len(actual_sequence)
for i in range(0,len(actual_sequence)):
    difference = actual_sequence[i]-predicted_sequence[i]
    if difference == 0.0:
        correct = correct + 1
print(float(correct)/float(total))
```