Project 2: Occupancy Detection

Sami Rodrigue

1 Introduction

For this project, I will be using Hidden Markov Models (HMM) to predict the occupancy of a room in the Occupancy Detection Data Set. The dataset contains a room's *temperature*, *humidity*, *light*, CO_2, *humidity ratio* and *occupancy* per minute.

The project has two main sections: Exploratory Analysis and Prediction. The first section explores the variables that will be used for the prediction task. The second section uses HMM to predict the occupancy of a room.

2 Exploratory Analysis

The target variable for the Occupancy Detection Data Set is discretized where 0 corresponds to not occupied and 1 corresponds to occupied. I will utilize two different HMMs to predict the occupancy, each one using a different variable from the dataset. The time resolution (Δt) for the models will be 15 minutes, and the values will be aggregated by taking their mean for each 15 minute time segment.

As the number of people increase in the room, the amount of CO_2 will also increase, hence it is a good candidate for predicting occupancy. Figure 1 shows how closely CO_2 and occupancy move together and this insight is confirmed by the correlation values presented in Table 1. The summary statistics for the aggregated data can be found in Table 2.

	Temperature	Humidity	Light	CO_2	Humidity Ratio	Occupancy
Temperature	1.000	-0.142	0.650	0.560	0.152	0.538
Humidity	-0.142	1.000	0.038	0.439	0.955	0.133
Light	0.650	0.038	1.000	0.664	0.230	0.907
CO_2	0.560	0.439	0.664	1.000	0.627	0.712
Humidity Ratio	0.152	0.955	0.230	0.627	1.000	0.300
Occupancy	0.538	0.133	0.907	0.712	0.300	1.000

Table 1: Correlation of the variables.

	Temperature	Humidity	Light	CO2	HumidityRatio	Occupancy
mean	20.6214	25.7467	120.2345	606.8795	0.0039	0.2141
std	1.0181	5.5433	191.6016	314.0136	0.0009	0.4010
25%	19.7511	20.1888	0.0000	438.8869	0.0031	0.0000
50%	20.4438	26.2198	0.0000	453.7434	0.0038	0.0000
75%	21.3553	30.5542	256.6672	644.7545	0.0044	0.0000
max	23.1256	39.0095	637.4688	2016.1250	0.0064	1.0000

Table 2: Summary statistics of the variables.

The HMM that uses CO_2 will emit 3 different values. If the CO_2 is less than 450 ppm, then it will emit 0. If the CO_2 is between 450 ppm and 650 then it will emit 1. Else it will emit 2. Hence, the emission model will be a 2 by 3 matrix.

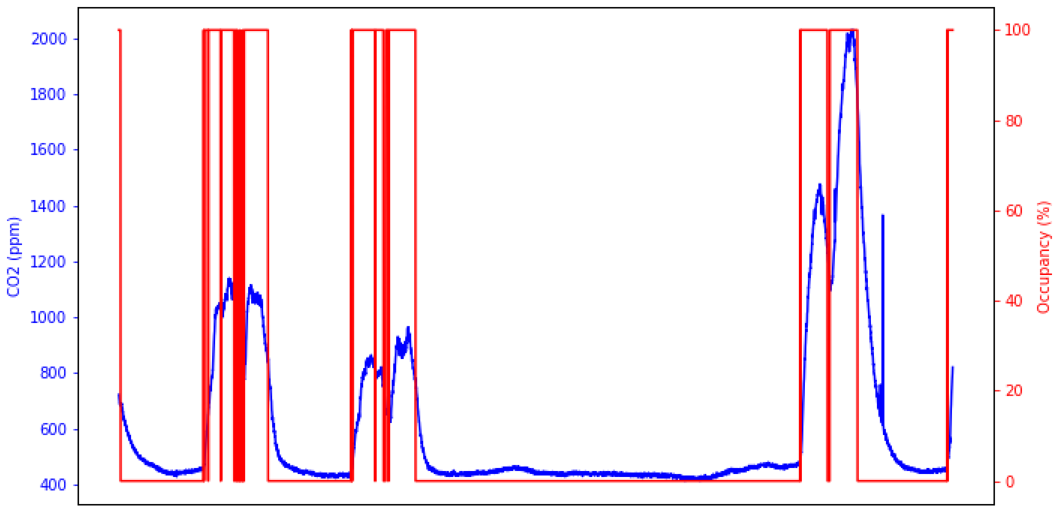

Figure 1: The CO_2 amount and the occupancy of the room over time.

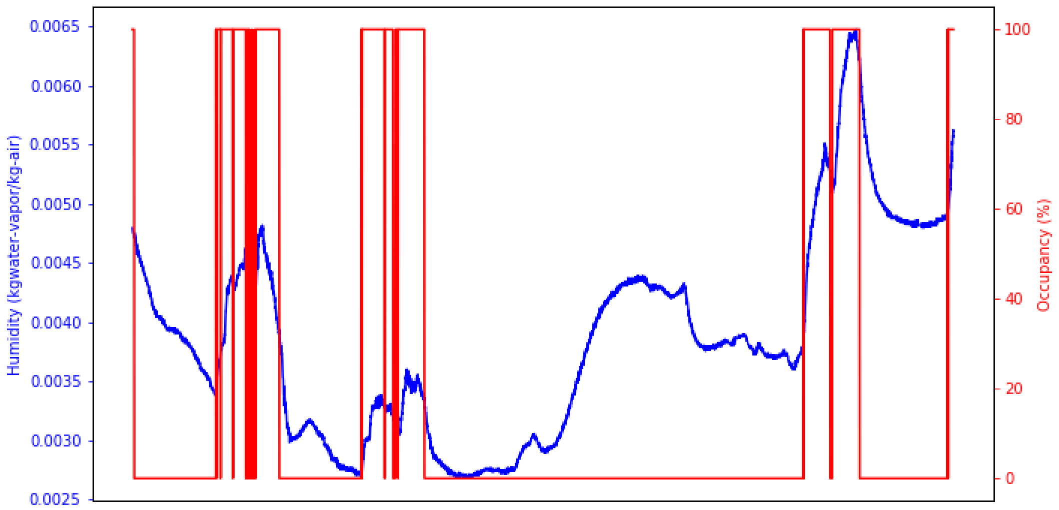

Figure 2: The humidity ratio and the occupancy of the room over time.

Table 1 indicates that humidity ratio has a high correlation with occupancy. Hence, I will pick this variable for my other model. As expected, Figure 2 shows humidity ratio and occupancy move closely together. This HMM will also emit 3 values. If the humidity ratio is less than 0.0037, then it will emit 0. If the humidity ratio is between 0.0037 and 0.0044 then it will emit 1. Else it will emit 2. Hence, the emission model will be a 2 by 3 matrix.

3 Solution Method

3.1 Methodology

For this project, I have used Hidden Markov Model (HMM) to identify the occupancy of the room. In a HMM, we can only measure discrete observations sampled at equally-spaced time intervals and they are emitted by N distinct states. For our project, $N = 2$ since we want to predict whether a room will be occupied without actually using this parameter.

The transition probabilities between the states follow the first order Markov property: the probability of observing a state at time t is only dependent on system's previous state at t-1. The transition probability is represented as $a_t^{i,j} = p(h_t = i|h_{t-1} = j), i, j \in N$ All transition probabilities are collected in a $N \times N$ transition matrix, which is denoted by A. The transition probabilities are dependent on the previous state except when t=1. This special case is called the initial state probability and it is represented as $\Pi = p(h_1 = i), i \in N$

Let us represent each observation at time t as o_t, which takes a distinct value from M where $M \in \{0, 1, 2\}$ since I have discretized the input into 3 different values. Let us represent the probability of observing an observation at time t as $b_t^{i,j} = p(o_t = i|h_t = j), i \in M, j \in N$

All emission probabilities are collected in a $N \times M$ emission matrix, which we denote by B. All of the parameters introduced in this section can be presented in the following compact notation $\lambda = (A, B, \Pi)$.

3.2 Experimentation

The Occupancy dataset is already divided into a testing and a training set. I have built two models where one uses CO_2 and the other uses humidity ratio as their observation $(\lambda_{CO_2}, \lambda_{HR})$. I have used the Baum-Welch algorithm to learn the parameters of each model and the parameters are randomly initialized. Once the features are learned, I have used the training set (aggregated with the same time resolution) to test their performance. I have used the Viterbi algorithm to retrieve the most likely sequence of hidden states. The occupancy values are rounded to the closest to compare them with the output of the Viterbi algorithm.

4 Results

I have used the confusion matrix and performance values related to it to evaluate my models and they can be found in Tables 3, 4 and 5. When using aggregated data, CO_2 is a better indicator then humidity ratio to predict a room's occupancy.

	1	0
1	60	19
0	5	94

Table 3: Confusion matrix of λ_{CO_2}

	1	0
1	62	51
0	3	62

Table 4: Confusion matrix of λ_{HR}

In %	Precision	Recall	Accuracy
λ_{CO_2}	75.95	92.31	87.5
λ_{HR}	54.87	95.38	69.66

Table 5: Results (Rounded to closest 2nd digit)

5 Appendix

This section includes code for the visualizations and the analysis

5.1 Code for Exploration

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import datetime

data = pd.read_csv('datatraining.txt')
data= data.set_index('date')
data.index = pd.to_datetime(data.index)

# descriptive stats
data.describe()
print(data.corr().round(3).to_latex())

# Vis1
fig, ax = plt.subplots(figsize=(10,5))

ax.plot_date(y= data['CO2'], x = data.index, fmt = '-', color='b')
ax.set_ylabel('CO2␣(ppm)', color='b')
ax.tick_params(axis='y', colors = 'b')
ax.tick_params(
    axis='x',
    which='both',
    bottom='off',
    top='off',
    labelbottom='off')

ax2 = ax.twinx()
ax2.plot_date(y= data['Occupancy']*100, x = data.index, fmt = '-', color='r')
ax2.set_ylabel('Occupancy␣(%)', color='r')
ax2.tick_params(axis='y', colors = 'r')
ax2.tick_params(
    axis='x',
    which='both',
    bottom='off',
    top='off',
    labelbottom='off')

fig.tight_layout()
plt.show()

# Vis2
fig, ax = plt.subplots(figsize=(10,5))

ax.plot_date(y= data['HumidityRatio'], x = data.index, fmt = '-', color='b')
ax.set_ylabel('Humidity␣(kgwater-vapor/kg-air)', color='b')
ax.tick_params(axis='y', colors = 'b')
ax.tick_params(
    axis='x',
    which='both',
    bottom='off',
    top='off',
    labelbottom='off')

ax2 = ax.twinx()
ax2.plot_date(y= data['Occupancy']*100, x = data.index, fmt = '-', color='r')
ax2.set_ylabel('Occupancy␣(%)', color='r')
ax2.tick_params(axis='y', colors = 'r')
ax2.tick_params(
    axis='x',
    which='both',
    bottom='off',
    top='off',
    labelbottom='off')

fig.tight_layout()
plt.show()
```

```
# aggregate dataset based on defined time resolution and calculate descriptive
    stats
newRes = data.resample('15T').mean()
newRes.describe()
print(newRes.describe().loc[['mean', 'std', '25%', '50%', '75%', 'max'],:].round(4)
    .to_latex())

# Discretize continuous variables as describe in the report
newRes['discreteCO2'] = np.where(newRes.CO2 < 450, 0, np.where(newRes.CO2 < 650, 1,
     2))
newRes['discreteHumidRatio'] = np.where(newRes.HumidityRatio < 0.0037, 0, np.where(
    newRes.HumidityRatio < 0.0044, 1, 2))
```

5.2 Code for Prediction

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import datetime

class HMM:
    def __init__(self, states, observations, pi=None, transition=None, emission=
        None):
        self.states = states
        self.pi = pi
        self.trans = transition
        self.emis = emission
        self.obs = observations
        self.initCalc()
        self.n = None

    def initCalc(self):
        if self.pi is None:
            self.pi = np.repeat(1/self.states, self.states)
        if self.trans is None:
            transmat = np.random.randint(low=1, high = 2000, size=(self.states,self
                .states))
            self.trans = transmat/transmat.sum(axis=1, keepdims=True)
        if self.emis is None:
            emissionprob = np.random.randint(low=1, high = 2000, size=(self.states,
                len(np.unique(self.obs))))
            self.emis = emissionprob/emissionprob.sum(axis=1, keepdims=True)

    def normalization(self, step=1):
        """ Normalization constant required for backward algorithm and gamma value
        """
        n = np.ones(len(self.obs))
        alpha = self.forwardUnnorm()

        n[0] = 1 / sum(alpha[:,0])

        for t in range(step, len(n), step):
            n[t] = 1 / (sum(alpha[:,t]) * np.prod([n[0:t]]))

        return(n)

    def forward(self):
        """ Return normalized alpha vales of the observation sequence
        """
        alpha = np.zeros((self.states, len(self.obs)))

        # initilization with normalization
        alpha[:,0] = self.pi * self.emis[:,self.obs[0]]
        alpha[:,0] = alpha[:,0] / sum(alpha[:,0])

        # t > 1 with normalization
        for t in range(1, len(self.obs)):
            for s in range(self.states):
                start = sum([alpha[i,t-1] * self.trans[i,s] for i in range(self.
                    states)])
                alpha[s,t] = start * self.emis[s, self.obs[t]]
            alpha[:,t] = alpha[:,t] / sum(alpha[:,t])
        return(alpha)

    def forwardUnnorm(self):
        """ Return alpha vales of the observation sequence
        """
        alpha = np.zeros((self.states, len(self.obs)))

        # initilization with normalization
        alpha[:,0] = self.pi * self.emis[:,self.obs[0]]

        # t > 1 with normalization
        for t in range(1, len(self.obs)):
            for s in range(self.states):
                start = sum([alpha[i,t-1] * self.trans[i,s] for i in range(self.
                    states)])
                alpha[s,t] = start * self.emis[s, self.obs[t]]

        return(alpha)

    def backward(self):
        """ Return normalized beta vales of the observation sequence
        """
        self.n = self.normalization()
        beta = np.zeros((self.states, len(self.obs)))

        # initilization
        beta[:,len(self.obs)-1] = 1

        # t < T with normalization
        for t in range(len(self.obs)-2, -1, -1):
            for s in range(self.states):
                beta[s,t] = self.n[t+1] * sum(self.trans[s,:] * self.emis[:, self.
                    obs[t+1]] * beta[:,t+1])

        return(beta)

    def backwardUnnorm(self):
        """ Return beta vales of the observation sequence
        """
        beta = np.zeros((self.states, len(self.obs)))

        # initilization
        beta[:,len(self.obs)-1] = 1

        # t < T with normalization
        for t in range(len(self.obs)-2, -1, -1):
            for s in range(self.states):
                beta[s,t] = sum(self.trans[s,:] * self.emis[:, self.obs[t+1]] *
                    beta[:,t+1])

        return(beta)

    def BaumWelch(self, iteration = 10):
        count = len(self.obs)

        for _ in range(iteration):
            # E-Step
            alpha = self.forward()
            beta = self.backward()

            # M-Step
            trans = np.zeros((self.states, self.states))
            emis = np.zeros((self.states, self.emis.shape[1]))

            pi = np.array([self.gamma(alpha, beta, i, 0) for i in range(self.states
                )])

            for i in range(self.states):
                for j in range(self.states):
                    num = sum([self.zeta(alpha, beta, i, j, t) for t in range(count
                        -1)])
                    denom = sum([self.gamma(alpha, beta, i, t) for t in range(count
                        -1)])
                    trans[i,j] =  num /denom

            for i in range(self.states):
                for j in range(self.emis.shape[1]):
                    num = sum([self.gamma(alpha, beta, i, t) *
                               np.where(self.obs[t] == range(self.emis.shape[1])[j
                                   ], 1, 0) for t in range(count)])
                    denom = sum([self.gamma(alpha, beta, i, t) for t in range(count
                        )])
                    emis[i,j] = num/denom

            self.trans = trans
            self.emis = emis
            self.pi = pi

    def Viterbi(self):
        count = len(self.obs)

        #init
        delta = np.zeros((self.states, count))
        psi = np.zeros((self.states, count))
        output = np.zeros(count)

        delta[:,0] = self.pi * self.emis[:,self.obs[0]]

        # recurse
        for t in range(1, count):
            for j in range(self.states):
                delta[j, t] = max([delta[i,t-1] * self.trans[i,j] for i in range(
                    self.states)]) * self.emis[j, self.obs[t]]
                psi[j, t] = np.argmax([delta[i,t-1] * self.trans[i,j] for i in
                    range(self.states)])

        # termination
        p = max([delta[i,count-1] for i in range(self.states)])
        q = np.argmax([delta[i,count-1] for i in range(self.states)])

        # backtracking
        output[count-1] = q
        for t in range(count-2, -1, -1):
            output[t] = psi[int(output[t+1]), t+1]

        return output

    def zeta(self, alpha, beta, i, j, t):
        num = alpha[i, t] * self.trans[i,j] * self.emis[j, self.obs[t+1]] * beta[j,
             t+1]
        denom = 0
        for k in range(self.states):
            for l in range(self.states):
                denom += alpha[k, t] * self.trans[k,l] * self.emis[l, self.obs[t
                    +1]] * beta[l, t+1]
        return (num / denom)

    def gamma(self, alpha, beta, i, t):
        num = alpha[i,t] * beta[i,t]
        denom = np.sum(alpha[:,t] * beta[:,t])
        return (num/denom)

data = pd.read_csv('datatraining.txt')
data= data.set_index('date')
data.index = pd.to_datetime(data.index)

newRes = data.resample('15T').mean()
newRes['discreteCO2'] = np.where(newRes.CO2 < 450, 0, np.where(newRes.CO2 < 650, 1,
     2))
newRes['discreteHumidRatio'] = np.where(newRes.HumidityRatio < 0.0037, 0, np.where(
    newRes.HumidityRatio < 0.0044, 1, 2))

# Learn parameters from the training data
np.random.seed(19)
data1 = newRes.discreteCO2.values
```

```
model1 = HMM(states= 2, observations = data1)
model1.BaumWelch(iteration =12)

data2 = newRes.discreteHumidRatio.values
model2 = HMM(states= 2, observations = data2)
model2.BaumWelch(iteration =12)

# Load test data
test = pd.read_csv('datatest.txt')
test= test.set_index('date')
test.index = pd.to_datetime(test.index)

test2 = test.resample('15T').mean()
test2['discreteCO2'] = np.where(test2.CO2 < 450, 0, np.where(test2.CO2 < 650, 1, 2)
    )
test2['discreteHumidRatio'] = np.where(test2.HumidityRatio < 0.0037, 0, np.where(
    test2.HumidityRatio < 0.0044, 1, 2))
target = np.round(test2.Occupancy.values)

# label test data by using Viterbi
model1.obs = test2.discreteCO2.values
pred1= model1.Viterbi()

model2.obs = test2.discreteHumidRatio.values
pred2= model2.Viterbi()

# Results
def conf(prediction, actual):
    TP = 0
    TN = 0
    FP = 0
    FN = 0
    for i in range(len(prediction)):
        if (prediction[i] == 1) & (actual[i] == 1):
            TP += 1
        elif (prediction[i] == 0) & (actual[i] == 0):
            TN += 1
        elif (prediction[i] == 1) & (actual[i] == 0):
            FP += 1
        else:
            FN += 1
    return(TP, TN, FP, FN)

tp1, tn1, fp1, fn1 = conf(pred1, target)
tp2, tn2, fp2, fn2 = conf(pred2, target)

precision1 = tp1/(tp1+fp1)
precision2 = tp2/(tp2+fp2)

recall1 = tp1/(tp1+fn1)
recall2 = tp2/(tp2+fn2)

acc1= (tp1 + tn1)/(tp1+ tn1+ fp1+ fn2)
acc2= (tp2 + tn2)/(tp2+ tn2+ fp2+ fn2)
```