YG390_lab-week6-student

October 21, 2019

1 DS-GA 3001.001 Special Topics in Data Science: Probabilistic Time Series Analysis

2 Week 6 Hidden Markov Model

[3]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import math
import random
import pylab
from collections import Counter
import time
```

2.1 Data

Load the Wall Street Journal POS dataset. Transform them into indices.

[4]:
```
train_dir = "../../data/en-wsj-train.pos"
test_dir = "../../data/en-wsj-dev.pos"

def load_pos_data(data_dir, word_indexer=None, label_indexer=None, top_k=20000):
    """
    Function that loads the data
    """
    with open(data_dir, "r") as f:
        # load data
        content = f.readlines()
        intermediate_X = []
        intermediate_y = []
        current_X = []
        current_y = []
        vocab_counter = Counter()
        label_set = set()
        for line in content:
            tokens = line.replace("\n", "").replace("$", "").split("\t")
            # !!! if problems use line below
            # tokens = line.replace("/n", "").replace("$", "").split("/t")
            if len(tokens) <= 1:
                intermediate_X.append(current_X)
                intermediate_y.append(current_y)
                current_X = []
                current_y = []
            elif len(tokens[1]) > 0:
                vocab_counter[tokens[0]]+=1
                label_set.add(tokens[1])
                current_X.append(tokens[0].lower())
                current_y.append(tokens[1])
        # index data
        top_k_words = vocab_counter.most_common(top_k)
        # 0 is reserved for unknown words
        word_indexer = word_indexer if word_indexer is not None else␣
↪dict([(top_k_words[i][0], i+1) for i in range(len(top_k_words))])
        word_indexer["UNK"] = 0
        label_indexer = label_indexer if label_indexer is not None else␣
↪dict([(label, i) for i, label in enumerate(label_set)])
        output_X = []
        output_y = []
        current_X = []
        current_y = []

        for i in range(len(intermediate_X)):
            for j in range(len(intermediate_X[i])):
                if intermediate_X[i][j] in word_indexer:
                    current_X.append(word_indexer[intermediate_X[i][j]])
                else:
                    current_X.append(0)
                current_y.append(label_indexer[intermediate_y[i][j]])
            # populate holders
            output_X.append(current_X)
            output_y.append(current_y)
            # reset current holder
            current_X = []
            current_y = []
        return output_X, output_y, label_indexer, word_indexer, {v: k for k, v␣
↪in label_indexer.items()}, {v: k for k, v in word_indexer.items()}

def reconstruct_sequence(idx_list, lookup):
    """
    Function that reconstructs a sequence from index
    """
    return [lookup[x] for x in idx_list]
```

[5]:
```
train_X, train_z, label_indexer, word_indexer, label_lookup, vocab_lookup =␣
 ↪load_pos_data(train_dir)
test_X, test_z, _, _, _, _ = load_pos_data(test_dir, word_indexer=word_indexer,␣
 ↪label_indexer=label_indexer)
```

2.1.1 data description

data are excerpts from the wall street journal

observations are words so that sentences form time series, the observation model is multinomial

we want to extract a latent space that characterizes the grammatical structure of sentences, the dataset is already labeled (for each observed word we are given the latent state which is the grammatical category the word belongs to)

more about the meaning of the latent space is here: https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_

we have the following variables ‘train_X’ = a list of lists, each list is a sentence with numbers representing words (observations)

‘word_indexer’ and ’vocab_lookup go from words to numbers, length 20001

‘label_indexer’ and’label_lookup’ go from category label to numbers, length 44

We will fit a HMM where each observed word is attributed to a category (“current state”).

[18]:
```
ii = 7
print("example training sentence: \n\n", reconstruct_sequence(train_X[ii],␣
 ↪vocab_lookup))
print("example's categories: \n\n", reconstruct_sequence(train_z[7],␣
 ↪label_lookup))
```

```
example training sentence:

 ['a', 'record', 'date', 'has', "n't", 'been', 'set', '.']
example's categories:

 ['DT', 'NN', 'NN', 'VBZ', 'RB', 'VBN', 'VBN', '.']
```

[19]:
```
print(len(train_X), ' training sentences')
print(len(test_X), 'testing sentences')
```

```
39815  training sentences
1700 testing sentences
```

2.2 Hidden Markov Model

observed states $X = \{x_1, x_2, ..., x_N\}$ and $x_n = h \in \{h = 1, ..., H\}$

observed variable given the latent could be for instance gaussian distributed where the latent state defines which gaussian they are from - in the case of our dataset above the observed variables are multinomial

latent states $Z = \{z_1, z_2, ..., z_N\}$ and $z_n \in \{1, ..., K\}$, so a discrete multinomial variable

Transition probability matrix: $A \in R^{KxK}$ where $A_{i,j} = p(z_n = j|z_{n-1} = i)$ and rows sum to 1

Emission probability matrix: $C \in R^{KxH}$ where $C_{i,h} = C_i(x_n = h) = p(x_n = h|z_n = i)$ and rows sum to 1

Initial State Probability: $\pi \in R^K$ where we will set the initial state to one specified category so that $\pi_i = p(z_1 = i)$

special cases of HMM models through constraint on transition probability matrix A

- slow changes and fast changes
- directional, eg left-to-right HMM
- assume latent states have a natural order and make large jumps unlikely

any entry of A that is set to 0 in the initialization will stay 0 during the EM updates

2.2.1 Likelihood (the forward algorithm)

given parameters $\theta = \{A, C, \pi\}$ and observation sequence X find the likelihood $p(X|\theta)$

compute the probability of being in a state j after seeing the first n observations, denote that probability as $\alpha_n(j)$

$$\alpha_n(j) = P(x_1, ..., x_n, z_n = j) = P(x_n|z_n = j) \sum_{i=1}^{K} \alpha_{n-1} P(z_n = j|z_{n-1} = i) =$$

$$C_j(x_n) \sum_{i=1}^{K} \alpha_{n-1}(i) A_{ij}$$

with initial condition

$$\alpha_1(i) = P(x_1|z_1 = i)P(z_1 = i) = \pi_i C_i(x_1)$$

so that

$$P(X|\theta) = \sum_{i=1}^{K} \alpha_N(i)$$

implementation caveat: introduce scaling to avoid numerical problems due to small probability values (see lecture or Bishop)

there is the equivalent for the other direction, called the Backward Algorithm, that uses

$$\beta_n(j) = p(x_{n+1}, ...x_N|z_n = j)$$

2.2.2 Viterbi Algorithm (max sum algorithm)

Aim: inference - estimate the latent state sequence, assume parameters are given

There are exponentially many possible chains of latent spaces in a sequence. However, let's assume we are given the probabilities up to n we can infer the probabilities for the next time step $n+1$. If we work our way forward recursively in this way, we can dramatically decrease the computational cost. Essentially, we only consider the path with the highest probability at every time point for each state.

$v_n(j)$ is the probability that we are in state j after seeing the first n observations and passing through the most probable latent state sequence

- initialization:
$$v_1(j) = \pi_j c_j(x_1)$$

for every time point n and every possible latent state $j = 1, ..., K$

- update:
$$v_n(j) = c_j(x_n) max_{i=1}^{K} v_{n-1}(i) a_{ij}$$

- store best state
$$b_n(j) = argmax_{i=1}^{K} v_{n-1}(i) a_{ij} c_j(x_n)$$

track the most likely sequence starting at the last time point

for implementation use the log form instead

2.2.3 parameter learning

in our dataset the latent states Z are given (E step of EM already solved), therefore we can update the parameters simply as follows:

$$\hat{a}_{ij} = \frac{num\ state\ transitions\ from\ i\ to\ j}{num\ state\ transitions\ from\ i}$$

$$\hat{c}_{ih} = \frac{num\ of\ times\ state\ i\ emits\ h}{num\ state\ i}$$

$$\hat{\pi}_i = \frac{num\ of\ chains\ start\ with\ i}{total\ num\ of\ chains}$$

2.3 HMM Implementation

In this part, you will implement the Hidden Markov Model with following methods:

- decode_single_chain: E-step use the Viterbi Algorithm to find the most probable sequence of states.

- sample: given an initial state and the number of steps, returns a sequence of sampled states and observations.

[10]:
```
def percentage_agree(x, z):
    """
    Function that shows the % of agreement among two list
    """
    assert len(x)==len(z)
    return float(np.sum(np.array(x)==np.array(z)))/len(x)

class MyHMM:
    def __init__(self, num_unique_states, num_observations):
        """
        Constructor
        @param num_unique_states: # of unique states (POS Tags)
        @param num_observations: # of unique observations (words)
        """
        self.num_unique_states = num_unique_states
        self.num_observations = num_observations
        self.transition_matrix = np.zeros((num_unique_states,␣
 ↪num_unique_states))
        self.emission_matrix = np.zeros((num_unique_states, num_observations))
        self.initial_states_vector = np.zeros(num_unique_states)

    def fit(self, X, z):
        """
        Method that fits the model.
        @param X: array-like with dimension [# of examples, # of length]
        @param z: array-like with dimension [# of examples, # of length]
        """
        # populate holders
        for i in range(len(z)):
            for j in range(len(z[i])):
                # update initial state probability
                if j == 0:
                    self.initial_states_vector[z[i][j]] += 1
                # update transition matrix
                if j < len(z[i])-1:
                    self.transition_matrix[z[i][j], z[i][j+1]] += 1
                # update emission matrix
                self.emission_matrix[z[i][j], X[i][j]] += 1
        # normalization
        self.initial_states_vector += 1e-10
        self.initial_states_vector /= np.sum(self.initial_states_vector)

        self.transition_matrix += 1e-10
        row_sums_transition_matrix = np.sum(self.transition_matrix, axis=1)
        self.transition_matrix /= row_sums_transition_matrix[:, np.newaxis]

        self.emission_matrix += 1e-10
        row_sums_emission_matrix = np.sum(self.emission_matrix, axis=1)
        self.emission_matrix /= row_sums_emission_matrix[:, np.newaxis]

    def decode_single_chain(self, x):
        """
        Auxiliary method that uses Viterbi on single chain
        @param X: array-like with dimension [ # of length]
        @return z: array-like with dimension [# of length]
        """
        # init holders
        z = []
        V = np.zeros( (len(x), self.num_unique_states) )
        best_states = np.zeros( (len(x), self.num_unique_states) )

        ##########################################
        # TODO: implement the Viterbi algorithm #
        ##########################################
        # w_1 = log(P(z_1)) + log P(x_1|z_1)
        # w_{i+1} = log(P(x_{i+1}|z_{i+1})) + arg max_{z_i} {log P(z_{i+1} |␣
↪z_i) + w_i}

        for t in range(len(x)):
            for s in range(self.num_unique_states):
                if t == 0:
                    prev_V = np.log(self.initial_states_vector)
                else:
                    prev_V = V[t-1]
                xx = prev_V +  np.log(self.transition_matrix[:,s]) + np.
↪log(self.emission_matrix[s, x[t]])
                largest_probability_state_index = np.argmax(xx)
                V[t, s] = xx[largest_probability_state_index]
                best_states[t, s] = int(largest_probability_state_index)

        optimal_state = int(np.argmax(V[len(x)-1, :]))
        z.append(optimal_state)
        for t in range(len(x)-1, 0, -1):
            optimal_state = int(best_states[t, optimal_state])
            z.append(optimal_state)

        return list(reversed(z))
        #return np.zeros(len(x))

    def decode(self, X):
        """
        Method that performs the Viterbi the model.
        @param X: array-like with dimension [# of examples, # of length]
        @return z: array-like with dimension [# of examples, # of length]
        """
        return [self.decode_single_chain(sample) for sample in X]

    def sample(self, n_step, initial_state, seed=0):
        """
        Method that given initial state and produces n_step states and␣
↪observations
        @param n_step: integer
        @param initial_state: an integer indicating the state
        """
        states = []
        observations = []
        current_state = initial_state

        np.random.seed(seed)

        ###############################
        # TODO: sample from the model #
        ###############################
        for i in range(n_step):
            transition_probabilities = self.transition_matrix[current_state]
            current_state = np.random.choice(np.arange(self.num_unique_states),␣
↪p=transition_probabilities)
            states.append(current_state)
            emission_probabilities = self.emission_matrix[current_state]
            current_observation = np.random.choice(np.arange(self.
↪num_observations), p=emission_probabilities)
            observations.append(current_observation)
        return states, observations
```

2.3.1 Learn an HMM

[11]:
```
num_states = len(label_indexer)
num_obs = len(word_indexer)
my_hmm = MyHMM(num_states, num_obs)
my_hmm.fit(train_X, train_z)
```

2.3.2 Use Viterbi to decode one single sequence

[12]:
```
i = 5 # decode the ith sentence in the testing dataset
res = my_hmm.decode_single_chain(test_X[i])
print("data: {0} \n\n pred: {1} \n\n true label: {2}".
 ↪format(reconstruct_sequence(test_X[i], vocab_lookup),
                                                    reconstruct_sequence(res,␣
 ↪label_lookup),
                                                                        ␣
 ↪reconstruct_sequence(test_z[i], label_lookup)))
```

```
data: ['this', 'financing', 'system', 'was', 'created', 'in', 'the', 'new',
'law', 'in', 'order', 'to', 'keep', 'the', 'bailout', 'spending', 'from',
'swelling', 'the', 'budget', 'deficit', '.']

 pred: ['DT', 'NN', 'NN', 'VBD', 'VBN', 'IN', 'DT', 'JJ', 'NN', 'IN', 'NN',
'TO', 'VB', 'DT', 'NN', 'NN', 'IN', 'VBG', 'DT', 'NN', 'NN', '.']

 true label: ['DT', 'NN', 'NN', 'VBD', 'VBN', 'IN', 'DT', 'JJ', 'NN', 'IN',
'NN', 'TO', 'VB', 'DT', 'NN', 'NN', 'IN', 'VBG', 'DT', 'NN', 'NN', '.']
```

decode for all

[14]:
```
start_time = time.time()
pred_train = my_hmm.decode(train_X[:1000])
print("takes {0} seconds".format(time.time() - start_time))
```

```
takes 6.258845090866089 seconds
```

[15]:
```
start_time = time.time()
pred_test = my_hmm.decode(test_X)
print("takes {0} seconds".format(time.time() - start_time))
```

```
takes 10.340965986251831 seconds
```

percent correct

[16]:
```
print('percent correct predicted in training: ', np.
 ↪mean([percentage_agree(pred_train[ii], train_z[ii]) for ii in␣
 ↪range(len(pred_train))]))
```

```
percent correct predicted in training:  0.9324096230230124
```

[17]:
```
print('percent correct predicted in testing: ', np.
 ↪mean([percentage_agree(pred_test[i], test_z[i]) for i in␣
 ↪range(len(pred_test))]))
```

```
percent correct predicted in testing:  0.9194595588167823
```

2.3.3 Sample

```
[19]: pos_tag, words = my_hmm.sample(10, label_indexer["NNP"])
      print('states: ',reconstruct_sequence(pos_tag, label_lookup))
      print('observed: ', reconstruct_sequence(words, vocab_lookup))
```

```
states:  ['NNP', '.', "''", 'NNP', ',', "''", 'PRP', 'VBZ', 'TO', 'VB']
observed:  ['UNK', '.', "''", 'association', ',', "''", 'him', 'is', 'to',
'achieve']
```

2.3.4 Please turn in the code before 23/10/2018 3:00 pm. Please name your notebook netid.ipynb.

2.3.5 Your work will be evaluated based on the code and plots. You don't need to write down your answers to these questions in the text blocks.

```
[ ]:
```