Basics of Hidden Markov Models

Machine Learning II 2019

2.1 Function for imputing missing characters (3p)

Define a function `impute_characters_1(x:str)` that imputes missing characters in words using belief propagation and outputs the last figure showing marginal posterior distributions for each location and maximum aposteriori estimate. Test it on four different words with missing characters.

Hint: For getting the maximum aposteriori estimate you have to generalise the appoach that was used in the previous notebook about Markov chains.

Grading: You get 1p for the treatment of marginal posterior probabilities and 2p for the maximum aposteriori estimate.

Solution

Consider the following Hidden Markov Model with:

- hidden states $S = \{s_1, s_2, ..., s_m\}$; in our case, S is the alphabet so each s_i is a character of the alphabet,
- sequence of observations $y_i \in S, i = 1, ..., n$, where n is the length of the sequence/word,
- initial probabilities $\boldsymbol{\beta}_{m\times 1}[\cdot]$ of states,
- transition probabilities $\boldsymbol{\alpha}_{m\times m}[\cdot,\cdot]$ between states,
- emission probabilities $\boldsymbol{\delta}_{m\times m}[\cdot,\cdot]$ between states and observations.

Each state always emits an observation but it can happen that we lose it. In that case, we assume it could have been any possible character s_i from the alphabet.

Let $X = (X_1, X_2, ..., X_n)$ be a random vector of hidden state sequences and let $x = (x_1, x_2, ..., x_n)$ be a particular instance of it, $x_i \in S$. Further, let

$$\mathsf{evidence} = \bigcup_{i=1}^{n} \mathsf{evidence}_i = \bigcup_{i=1}^{n} \{y_i \oplus \text{'}-\text{'}\},$$

where $\text{'}-\text{'}$ denotes a missing observation and $\oplus$ is an operator for exclusive disjunction (xor). Due to Markov properties and the fact that $\mathsf{evidence}_i$ are independent from each other, we have

$$\begin{aligned}
\Pr[\boldsymbol{x}|\mathsf{evidence}] &\propto \Pr[\mathsf{evidence}|\boldsymbol{x}] \cdot \Pr[\boldsymbol{x}] \\
&\propto \prod_{i=1}^{n} \Pr(\mathsf{evidence}_i|X_i = x_i) \prod_{i=1}^{n} \Pr(X_i = x_i|X_{i-1} = x_{i-1}) \\
&\propto \prod_{i=1}^{n} \boldsymbol{\delta}[x_i,\cdot]\lambda_{Y_i} \prod_{i=2}^{n} \boldsymbol{\alpha}[x_{i-1}, x_i] \cdot \boldsymbol{\beta}[x_1] \\
&\propto \prod_{i=2}^{n} \boldsymbol{\delta}[x_i,\cdot]\lambda_{Y_i}\boldsymbol{\alpha}[x_{i-1}, x_i] \cdot \boldsymbol{\delta}[x_1,\cdot]\lambda_{Y_1}\boldsymbol{\beta}[x_1],
\end{aligned}$$

where λ_{Y_i} is an $m \times 1$ vector of likelihoods of the i-th observation (if missing then a vector of ones, otherwise a dummy/one-hot vector).

We are interested in finding the sequence of hidden states that maximizes the probability above. It is infeasible to calculate the above probability for every possible sequence. (Why is it infeasible and how many different sequences are there in the HMM we defined above? Drop me an e-mail with an answer for 1 extra point.) A better idea would be to use the Viterbi algorithm. Staying true to the notation used for Markov chains, this boils down to finding

$$\pi_i^*(v) = \begin{cases} \max_u \left(\pi_{i-1}^*(u) \cdot \boldsymbol{\alpha}[u,v] \cdot \boldsymbol{\delta}[v,\cdot]\lambda_{Y_i}\right), & i \geq 2 \\ \max_v \left(\boldsymbol{\beta}[v] \cdot \boldsymbol{\delta}[v,\cdot]\lambda_{Y_1}\right), & i = 1 \end{cases}$$

and then keeping track of the corresponding maximal states u. Essentially, for every state v at timepoint t we find the most probable state $u \in S$ at the previous timepoint from where we arrived to v.

An example piece of code for finding the most probable hidden sequence:

```python
def max_a_posteriori_path(y, alpha, beta, delta, likelihood):
    n = len(y)
    max_prob = [None] * n
    previous_letter = [None] * n

    if y[0] == '-':
        max_prob[0] = beta
    else:
        max_prob[0] = beta * delta.dot(likelihood['y'][0])

    for i in range(n - 1):
        probs = np.outer(delta.dot(likelihood['y'][i+1]), max_prob[i]) * alpha.T
        previous_letter[i + 1] = probs.idxmax(axis = 1)
        max_prob[i + 1] = probs.max(axis = 1)

    max_path = [None] * n
    max_path[n - 1] = max_prob[n - 1].idxmax()
    for i in reversed(range(1, n)):
        max_path[i - 1] = previous_letter[i][max_path[i]]

    return max_path
```

2.2 Language detection with Hidden Markov Models (1p)

Use files `est_training_set.csv` and `eng_training_set.csv` in the directory `data` to learn model parameters α and β for homogenous Markow chains. Use the emission probabilities $\boldsymbol{\delta}$ specified above to get the full parametrisaion of HMM.

Put these parameters into the formal model to compute probabilities

$$p_1 = \Pr[word|\text{Estonian}]$$
$$p_2 = \Pr[word|\text{English}]$$

and then use Bayes formula

$$\Pr[\text{Estonian}|word] = \frac{\Pr[word|\text{Estonian}]\Pr[\text{Estonian}]}{\Pr[word]}$$

to guess the language of a word on test samples `est_test_set.csv` and `eng_test_set.csv`.

Why does the procedure work if you use maximum likelihood estimates for the parameters?

Solution

Here, we need to find the probability that either language "emitted" a specific word. We can write this as

$$\Pr[\mathsf{evidence}] = \sum_{\boldsymbol{x}} \Pr[\mathsf{evidence} \wedge \boldsymbol{x}] = \sum_{\boldsymbol{x}} \Pr[\mathsf{evidence}|\boldsymbol{x}] \Pr[\boldsymbol{x}],$$

where the sum is over all possible hidden state sequences $\boldsymbol{x}$. We already saw above how to find the probability terms inside the sum – $\Pr[\mathsf{evidence}|\boldsymbol{x}]$ and $\Pr[\boldsymbol{x}]$ – for any given hidden sequence $\boldsymbol{x}$. We could brute force over all the possible hidden state sequences but it is infeasible as mentioned above. A better idea would be to use the forward algorithm which actually results to pretty much the same code as we used to find the most probable hidden sequence.

Note that before

for every state v at timepoint t we found the most probable state $s \in S$ at the previous timepoint from where we arrived to v.

But now

for every state v at timepoint t we sum the probabilities that we arrived from states $s_i \in S$ $(i = 1, \ldots, m)$ at the previous timepoint.

An example piece of code for finding word probabilities:

```python
def word_probability(word, alpha, beta, delta, alphabet):
    if len(word) == 0:
        return 0

    emission_likelihood = Series(0, index = alphabet)
    emission_likelihood[word[0]] = 1
    prob = beta * delta.dot(emission_likelihood)

    for letter in word[1:]:
        emission_likelihood = Series(0, index = alphabet)
        emission_likelihood[letter] = 1
        prob = np.sum(np.outer(delta.dot(emission_likelihood), prob) * alpha.T,
            axis = 1)

    return np.sum(prob)
```

Finally, the procedure works even if we use maximum likelihood estimates because we allowed for typing errors in the emission probabilities.